Approved for public release; distribution is unlimited

AN EXPERT SYSTEM FOR HIGH LEVEL MOTION CONTROL FOR AN AUTONOMOUS MOBILE ROBOT

by

Robert William Fish
Lieutenant Commander, United States Navy
B.S.S.E., United States Naval Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1993

CDR Gary Hughes, Chairman,
Department of Computer Science

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION

1. TITLE (Include Security Classification)
AN EXPERT SYSTEM FOR HIGH LEVEL MOTION CONTROL FOR AN AUTONOMOUS MOBILE ROBOT (U)

2. PERSONAL AUTHOR(S)
ish, Robert William

3a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 07/91 TO 06/93	June 1993	106

6. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Expert Systems, Mobile Robots, Obstacle Avoidance, CLIPS

9. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Computer Science Department at the Naval Postgraduate School in Monterey, California performs research he control and operation of autonomous mobile robots. One such robot, Yamabico-11, is an excellent test platfo or the study of path planning and obstacle avoidance. The ability to operate in an area where unforeseen obstac re present, and still attain the specified goal, is a highly desirable behavior in an autonomous mobile robot. This t is takes a step in that direction by proposing and implementing an expert system for high level motion control of obot. The expert system combines basic path planning routines and advanced obstacle avoidance techniques to dir he robot as it performs the mission.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Yutaka Kanayama	22b. TELEPHONE (Include Area Code): (408) 656-2095	22c. OFFICE SYMBOL: CS/Ka

ABSTRACT

The Computer Science Department at the Naval Postgraduate School in Monterey, California performs research on the control and operation of autonomous mobile robots. One such robot, Yamabico-11, is an excellent test platform for the study of path planning and obstacle avoidance. The ability to operate in an area where unforeseen obstacles are present, and still attain the specified goal, is a highly desirable behavior in an autonomous mobile robot. This thesis takes a step in that direction by proposing and implementing an expert system for high level motion control of the robot. The expert system combines basic path planning routines and advanced obstacle avoidance techniques to direct the robot as it performs the mission.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

A mobile robot that operates in a completely known environment, one where there are no changes or surprises, can accomplish its mission without the ability to detect and avoid obstacles. All that is needed is a stored map of the unchanging environment and a good position keeping system. However, when an autonomous mobile robot shares a world with people, there will be situations that cannot be stored on a pre-drawn map. The true autonomous mobile robot must be able to sense the environment it is operating in, and be able to react to unforeseen situations that may be encountered while operating.

A common situation that may be encountered by a robot is the presence of one or more obstacles between the robot and its destination. Since the location of the obstacles cannot be predicted in advance, the robot must be able to sense the presence of the obstacle, and find a way to go around the obstacle, if space permits. An expert system is the best way to guide the robot around multiple obstacles. In addition, an expert system can provide behavior rules and environmental interaction criteria to allow a multi-goal mission to be planned and executed, thereby making the autonomous mobile robot even more independent than before.

B. OVERVIEW

The thesis research involves three main parts. First, the algorithm and supporting behavior rules that make up the expert system were developed, and are described in Chapter III. Second, the expert system was implemented in CLIPS on a Sun workstation as a modification to the Yamabico simulator. This work is discussed in Chapter V. Finally, the CLIPS rules and functions were translated to C and implemented on the autonomous mobile robot Yamabico-11, as discussed in Chapter VI.

C. RELATED WORK

Previous work on Yamabico-11 includes avoidance of a single rectangular obstacle encountered while following a straight line [Alex93].

Ongoing research into image recognition and evaluation will provide Yamabico-11 with the ability to extract more information about an obstacle than just its range. Developing a system to perform automated cartography of an unknown environment is the subject of additional research.

II. PROBLEM STATEMENT

A. AUTONOMOUS TRAVEL

The problem involves the development of behavior rules for an autonomous mobile robot to permit travel from an arbitrary start position to an arbitrary goal position in a partially-known, orthogonal environment. The environment is considered partially known in that the location and size of fixed objects, such as walls, is known and is available to the robot. The unknown part of the robot's world is the possible presence of obstacles that may block the robot's path, and must be avoided. The first requirement for avoidance behavior is the ability to sense the environment, specifically the presence of obstacles. The second requirement is a method for going around one or more obstacles, and continuing on to the goal.

B. ASSUMPTIONS

Since the behavior used to avoid an obstacle depends greatly on the ability of the sensors that detect the obstacle, the behavior rules developed are appropriate for the test vehicle: Yamabico-11. Yamabico-11 is equipped with 12 ultrasonic sonars capable of detecting various sizes of targets at ranges up to 410 centimeters [NASA91]. The narrow beamwidth of the sonars, along with the size and aspect requirements of the target, have forced the following simplifying assumptions to be made.

The robot is operating in a partially known, orthogonal world. The world is partially known in that the robot has information on the location of fixed walls, but has no information on the positions of doors, or the presence of obstacles in the world. The world is orthogonal in that all walls meet at right angles. The normal operating environment for Yamabico-11 is the fifth floor of Spanagel Hall, which adequately meets the above assumptions.

An obstacle is any object that is not part of the known world. All obstacles are orthogonal in shape, they are composed of combinations of rectangles. The size of the obstacle is sufficient to allow detection by the robot's sonar sensors. The placement of the obstacles is arbitrary, with the restriction that an axis of each obstacle is parallel to the walls of the known world. Due to sonar beamwidth limitations, any two obstacles are assumed to be spaced far enough apart to allow the robot to pass between them. Otherwise, a significant amount of maneuvering is required to verify that sufficient clearance exists.

Obstacles are assumed to be stationary while the robot is in the vicinity of the obstacle. However, if the robot leaves and then returns to the same area, the obstacle may be in the same position, a different position, or it may be gone.

The robot's path will consist of a series of lines that are orthogonal to the environment. This ensures that all obstacles will present the best possible aspect to the sonar sensors.

Figure 1 shows a sample world and a possible path that the robot might take to achieve the goal.

C. YAMABICO-11

Yamabico-11 is an autonomous mobile robot powered by two 12-volt batteries and is driven on two wheels by DC motors. These motors drive and steer the robot, while four spring-loaded caster wheels provide balance.

The master processor is a MC68020 32-bit microprocessor accompanied by a MC68881 floating point coprocessor. (This is the exact same CPU as a Sun-3 workstation). This processor has one megabyte of main memory and runs with a clock speed of 16MHz.

All programs on the robot are developed using a Sun 3/60 workstation and UNIX operating system. These programs are first compiled and then downloaded to the robot via a RS-232 link at a baud rate of 9600. The software system consists of a kernel and a user program. The kernel is approximately 82,000 bytes and only needs to be downloaded once during the course of a given experiment. The user's program can be modified and

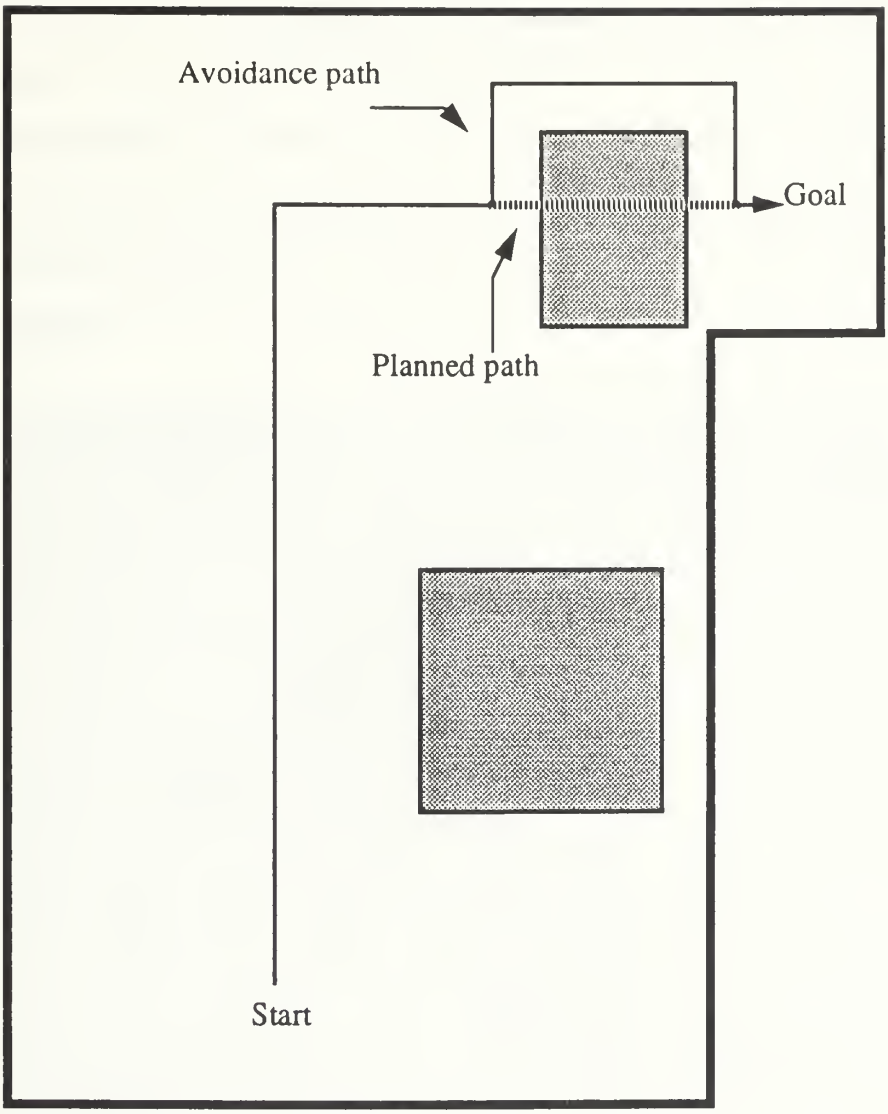


Figure 1 - Sample World with Two Obstacles

downloaded quickly to support rapid development. A laptop console is provided to accomplish command level communications to and from the user.

Twelve 40 kHz ultrasonic sensors are provided to allow the robot to sense its environment. The sonar subsystem is controlled by an 8748 micro-controller. Each sonar reading cycle takes approximately 24 milliseconds. [User93]

Motion and sonar commands are issued by the user in MML, the model-based mobile robot language. A path specification, such as a line, is passed to the robot. The robot then adjusts the curvature of its motion to follow the desired line. Transitions between successive paths are performed automatically by the robot. A summary of the MML motion and sonar commands used in the expert system is included as Appendix A.

Yamabico-11 is used as a test platform for research in path planning, obstacle avoidance, environment exploration, path tracking, and image understanding. Figure 2 shows a picture of Yamabico-11.

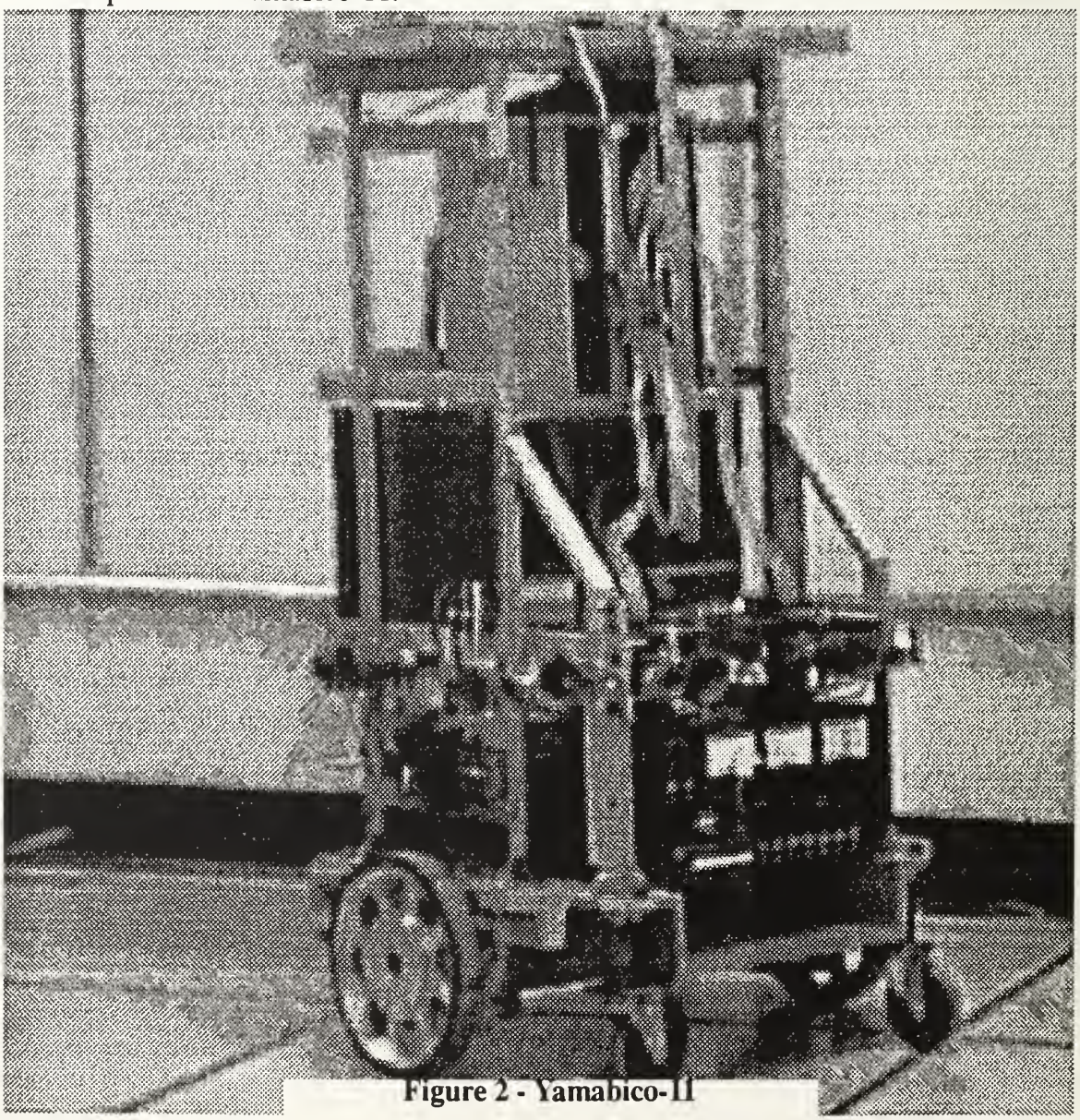


Figure 2 - Yamabico-11

III. SOLUTION

A. APPROACH

The critical part of the autonomous travel behavior is the successful avoidance of obstacles, since this requires dynamic planning with no guarantee that there exists a path around the obstacle.

In developing the avoidance behavior it is not sufficient to limit the robot to avoiding one obstacle at a time. The robot must be able to detect an obstacle, compute an avoidance path that will give the best chance of getting around the obstacle, and follow that path, while still watching for additional obstacles. Once the obstacle has been avoided, the robot may need to compute a new path to the goal, or simply regain the original path on the other side of the obstacle. There are other factors involved, such as proximity to the walls of the environment, and the size of the robot's odometry error.

The logical complexity involved in the described behavior prompted the development of an expert system to compute the paths for the robot to follow. The expert system operates as a supervisory control system for the robot, taking in position information and sensor data from the robot, and computing the course of action necessary to achieve the goal.

The basis for the operation of the expert system is the rules that "an expert" would use when faced with a similar situation. In this case, the expert can be anyone who has ever tried to walk in an area where boxes, desks, or other orthogonal obstacles are in the way. To bring the comparison down to the limitations of the robot, the person is assumed to know where he is going, and the size and shape of the room, but he has blinders on, so that he can see only a short distance with a narrow field of view. When an obstacle is encountered, one would normally turn to go around the obstacle in the direction that minimizes the extra distance traveled. However, with the blinders on, one cannot usually see the edges of the obstacle, so the best direction to turn may not be immediately obvious.

The expert system is composed of seven independent modules, divided according to function. Each module is responsible for information and decisions related to its function. A module may request information and decisions from other modules to aid in making its own decision. The modules are organized in a blackboard architecture, with most modules independently writing information and decisions to the blackboard, while other modules are responsible for interpreting and acting on information that is on the blackboard. The seven modules are the path planning module, the goal achievement module, the shortest path module, the wall avoidance module, the obstacle recognition module, the obstacle history module, and the odometry error module.

B. EXPERT SYSTEM

1. Main Algorithm

The top level algorithm for the expert system is based on the assumptions listed in Chapter II. The algorithm takes as input the current robot position and a list of one or more goals. The goals may be determined by the user or by some higher level strategic mission planner. A typical mission is mail delivery, with the sequence of goals representing the various offices on the floor that need mail. The last goal on the list will be the mail room, so that the robot returns for more mail to be delivered. The output of the function is a list of goals that were actually achieved; some goals may have been blocked by obstructing obstacles.

A pseudocode description of the main algorithm for the expert system is shown in Figure 3. Most of the variable types are consistent with those used in MML, others are appropriate to standard data structures. The major functions of the algorithm are described in the appropriate section below, depending on which module is responsible for each function.

```
procedure execute_mission (robot_position : configuration
                           mission_goals : in goal_list
                           goals_achieved : out goal_list) is

    configuration  :  start, goal;
    map            :  world_map;
    map_list       :  known_obstacles;
    boolean        :  goal_achieved, obstructing_obstacle, odometry_error_excessive,
                      cannot_reach_goal;

begin
    -- Start position is the first configuration in incoming list.
    start := robot_position;
    -- Loop until there are no more goals to achieve.
    while ( not is_empty (mission_goals)) loop
        goal := get_next_goal(mission_goals);
        -- Start a loop of planning, moving, and avoiding obstacles until goal_achieved.
        while (not goal_achieved) loop
            path_sequence = plan_path_to_goal (start, goal,world_map, known_obstacles)
            -- If path planner cannot find a safe path to the goal, the goal may be replaced
            -- with the next goal. See description of plan_path_to_goal.
            loop
                exit when  (goal_achieved or obstructing_obstacle or
                            odometry_error_excessive)
                follow_path (path_sequence);
            end loop;
            if (obstructing_obstacle) then
                -- Discard remainder of path, work on getting around obstacle.
                flush_rest_of_path (path_sequence);
                loop
                    exit when (not obstructing_obstacle or cannot_reach_goal);
                    path_sequence = avoid_obstacle (obstructing_obstacle, goal,
                                                    world_map, known_obstacles);
                    -- If there is no way around the obstacle, the goal may be replaced
                    -- with the next goal. See description of avoid_obstacle.
                    follow_path (path_sequence);
                end loop;
            end if;
            if (odometry_error_excessive) then
                resolve_odometry_error;
            end if;
        end loop;
        if (goal_achieved) then
            -- Goal has been achieved, add to goals_achieved, advance start position to goal.
            add_to_list(goals_achieved, goal);
            start := goal;
        end if;
    end loop;
end execute_mission;
```

Figure 3 - Expert System Main Algorithm

2. Path Planning Module

This module performs most of the work of the expert system. It is responsible for long and short range path planning, as well as reasoning about unachievable goals and other situations. The first task of the path planner is to plan a sequence of paths from the current robot position, which may be the starting position, to the goal. The second task is to issue sequential commands to avoid a newly detected obstacle. The last task is to resolve the case where the odometry error estimate has grown too large.

a. Computing a Path to the Goal

This is a complex process, involving several iterations of calculations, testing, and refining, until a safe path has been found from the current position to the goal. The desired path is the shortest overall distance to the goal that provides adequate clearance both from the known walls of the environment, and from the edges of previously detected obstacles that are assumed to still be in their last known positions.

The simplifying assumptions made for the case of Yamabico-11 preclude the choice of the true shortest path in most cases, since the lines used must be orthogonal to the world. Therefore, the final path sequence will consist of several intersecting lines that provide the necessary clearance around the known obstacles and walls. For more information on path planning, see [Brutz92] and [Kana92]. A description of the steps involved in function *plan_path_to_goal* follows.

Parameters passed to the function *plan_path_to_goal* are the starting and goal configurations, the map of the environment, and the maps of any previously detected obstacles. The output of the function is a sequence of lines that will safely achieve the goal, unless additional obstacles are encountered while following the path sequence.

The shortest path module provides the first major assist to the path planning module. It computes two lines to get to the goal. The first line normally begins at the start position and moves in the x direction until the goal configuration's x coordinate is reached.

The second line, perpendicular to the first, achieves the goal. While this path is by no means shortest, it allows for the sonar system limitations discussed in the system overview.

The path sequence must be checked for possible impact with environment walls. The wall location data is provided by the wall avoidance module. If the walls are too close to the path, the path is modified to avoid the walls. Possibly the order of execution of the two lines can be switched. Additional intermediate lines may be needed to allow for more complex environments.

The latest path sequence must be checked for possible impact with previously detected (and now known) obstacles. This information is the responsibility of the obstacle history module. If an obstacle is too close to the planned path sequence, then a decision must be made concerning the continued existence of the obstacle in the same location. The obstacle history module is called on to make this decision, since the rules and age data about previously detected obstacles are kept there. If the decision is that the obstacle is no longer in the same spot, then the path planner will plan the path as if the obstacle is not there. If the decision was wrong, the robot will eventually detect the obstacle and be forced to take avoidance action as described below.

If the obstacle history module decides that the obstacle is still in the same spot, then the prudent thing to do is to include the obstacle in the planning process, and compute a path that misses the obstacle, yet still achieves the goal. Several iterations may be necessary to get a clear path to the goal, avoiding the walls of the environment as well as existing obstacles. If the path planning module cannot find a safe path to the goal (obstructed by obstacles), then the process becomes more complicated. Three courses of action are possible: first, the robot can abandon the goal and go on to the next goal. This may be appropriate if the robot is on a multi-goal route, and the current goal is not unusually important right now. Second, the robot can abandon the entire mission and return to the start position, displaying a message to inform the user that the goal was not achieved and why. Finally, the robot can continue to try to achieve the goal. This involves moving toward the goal in the hope that the obstructing obstacle is no longer there, or has moved. In this

case, the output of the obstacle history module is ignored, even if there is a high probability that a previously detected obstacle is still there. The action selected depends mainly on the importance rating associated with the goal. If the goal achievement module posts a high importance rating for the goal, then the path planner will continue to try to achieve the goal. The end result will have the robot patrolling back and forth in front of the obstacles, waiting for one of the obstacles to move, perhaps using speech circuits to request someone to move an obstacle.

b. Avoiding a New Obstacle: Initiating Wall Following

A new obstacle in a position that obstructs the robot will require a new path to avoid it. The path planning module executes the function *avoid_obstacle* to calculate a new path sequence to avoid the obstacle. Since the problem assumptions specify orthogonal obstacles, the avoidance path starts with a line perpendicular to the current line. The robot will be commanded to follow the edge of the obstacle, "wall following", until the obstacle is no longer in the way.

The "best" direction of the initial turn depends on the location of walls, previously detected obstacles, and the location of the goal in relation to the robot's current heading. This data is shown as arguments to *avoid_obstacle*, as shown in Figure 3. If the goal is to the right of the robot's heading, and walls are not near, then the "best" direction to turn is right, and wall follow with the obstacle on the robot's left, until wall following termination criteria are met. If a wall of the environment is near the robot's current position, then the probability that a safe path exists between the obstacle and the wall is low. In this case, the "best" direction to turn is away from the wall.

The process of wall following around an obstacle involves two requirements. The first requirement is that the robot must correctly recognize and negotiate interior and exterior corners of the obstacle. These corners must be turned each time there is sufficient room to perform the turn safely, since the only way around the obstacle may be through a narrow opening. The wall of the obstacle may have variations in it that are technically

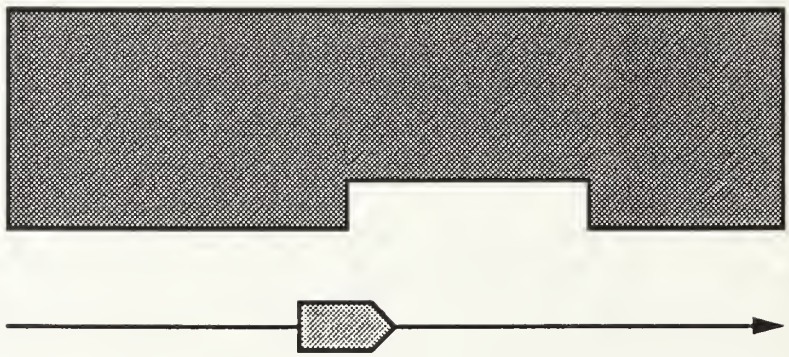
corners, however the robot does not need to follow these changes unless the change is large enough to be a possible opening. The amount of change required to initiate a turn depends on the size of the robot, its normal turning radius, and the desired distance away from the wall while wall following. Figure 4 (a) shows an example of wall variations that do not need to be followed. Figure 4 (b) shows a change in the wall that must be followed to find the opening and go around the obstacle.

The second requirement of wall following is that the robot must recognize when the obstacle extends to a wall of the environment, or extends to a point near the wall such that there is not enough room between the environment wall and the obstacle for the robot to safely operate. The minimum clearance required depends on the physical size of the robot, the odometry error, and its ability to turn around should the opening prove to be a dead end. For example, the minimum clearance for Yamabico-11 is 60 centimeters. This figure is based on a robot width of 54 centimeters and zero turning radius, since the robot can stop and back up if required.

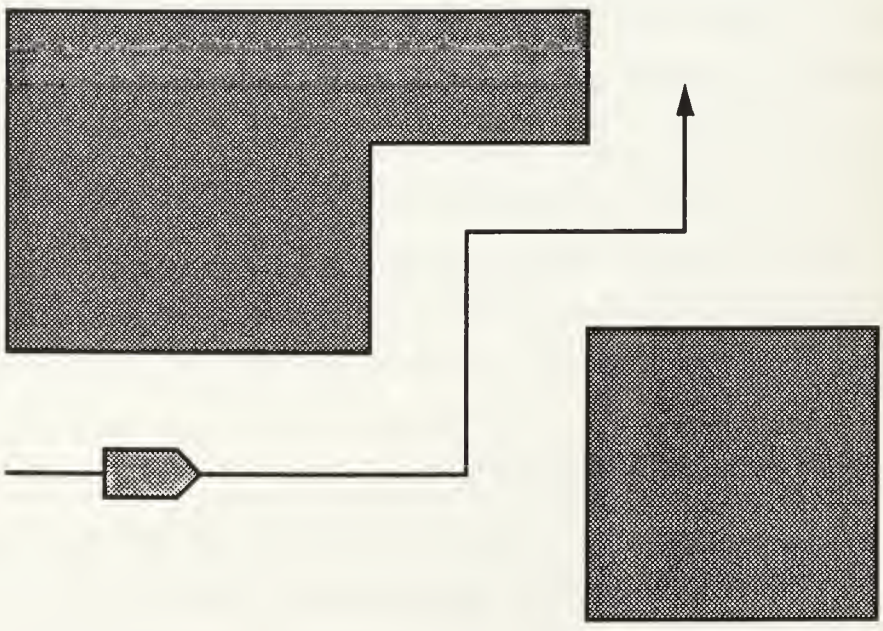
When an environment wall is reached, the robot must turn around and reverse the direction of wall following to try to get around the obstacle the other way. This maneuver is accomplished with the assistance of the wall avoidance module, which alerts the path planner when the robot's present course is approaching a wall of the environment. When it is clear that there is not enough room to get between the obstacle and the wall, the path planner commands the robot to reverse its course and wall follow with the obstacle on the robot's other side. Figure 5 shows an example of wall following reversal due to encountering an environment wall.

c. Termination of Wall Following

While there are only two requirements to perform the actual wall following process, there are very complex requirements to determine when to stop wall following and continue with the algorithm. From the human expert's perspective, the time to stop wall-following is obvious: when the obstacle is no longer obstructing a reasonable path to the



(a)



(b)

Figure 4 - Interior and Exterior Corners While Wall Following

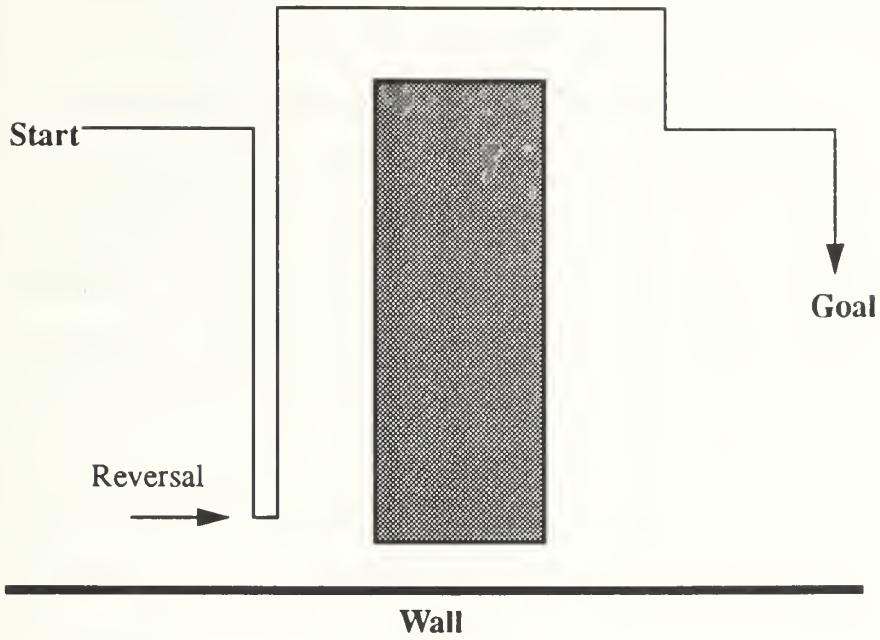


Figure 5 - Reversing the Direction of Wall Following

goal. However, translating these criteria into rules for the robot to use is difficult, considering the variety of environment and obstacle arrangements that are possible, even with the simplifying assumptions that have been made. An assortment of criteria are available for determining when to stop wall following and continue on towards the obstacle. Several of these criteria are examined below.

-- Count the number of corners completed while wall following. While this is a simple test to perform, it is not suitable for obstacles with an arbitrary number of corners, even if the obstacle is orthogonal. However, the fact that the robot must complete *at least one* exterior corner in order to get around an orthogonal obstacle is useful. It means that testing for termination of wall following may be necessary only at exterior corners, instead of continuously.

-- Wait until the robot's heading returns to the same direction as it was when wall following began. While this test works for obstacles with some interior corners, it can be defeated by obstacles with box canyons.

-- Wait until the robot reaches a point closer to the goal. This method has been effective in an obstacle avoidance expert system that was based on a grid environment, as described in Chapter V. However, the criteria is not directly practical when applied to a robot that has a ten millisecond motion control interval. While wall following, it is possible to move closer to the goal and still be obstructed by the obstacle. Therefore, the robot would terminate wall following, and immediately be forced to start wall following again, because the obstacle is still in the way.

-- A combination of the above criteria is possible. One such combination that works well is to compute the distance to the goal after negotiating each exterior corner of the obstacle. This system will eventually get around the obstacle, but the subsequent paths to the goal will have to be re-computed. Depending on the sophistication of the path planner, the new path may force the robot right back into a wall following mode around the same obstacle, perhaps in the opposite direction.

The criteria for termination of wall following for the expert system uses some of the above concepts. The test, though easily stated, requires some extensive testing and storage of data. After completing a turn at an exterior corner of the obstacle, the path planner checks to see if the line being followed intersects a segment of the originally planned path at a point farther along the path than where wall following began. This test requires that the line segments that made up the originally planned path be stored in a data structure, probably a list, for ease of searching. Routines must be used to test to see if the line that the robot is currently following intersects any of the segments in the list. If an intersection occurs, a check is made to make sure the intersection is closer to the goal, in terms of distance along the path, than the point where wall following began. A final check is required to verify that the intersection does not occur on the other side of a wall of the environment. Figure 6 shows two possible arrangements where the termination criteria for

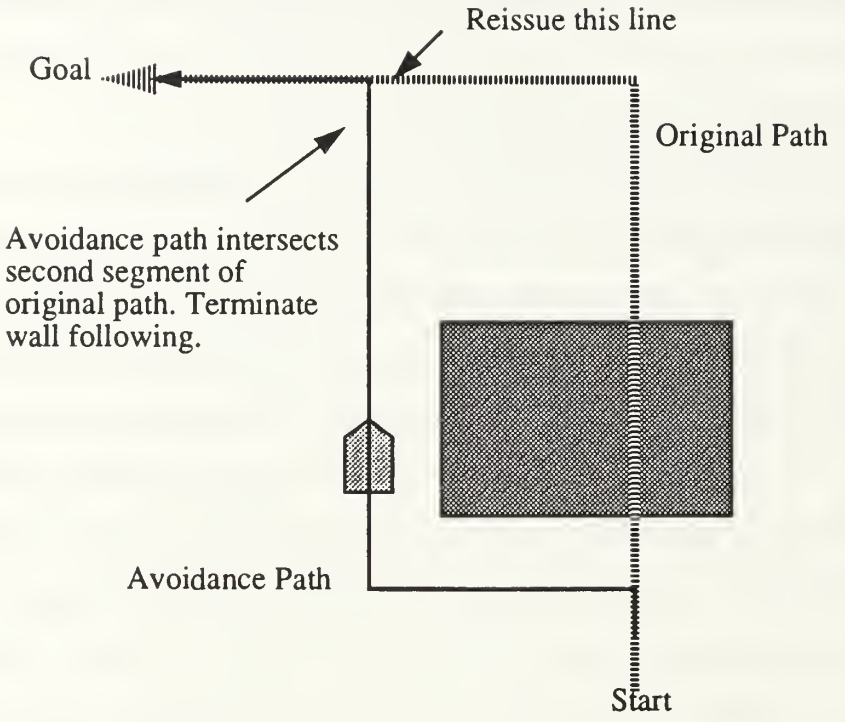
wall following are met. Figure 7 shows three path segment intersections that do not meet the criteria for termination of wall following. When wall following is terminated, the boolean *obstructing_obstacle* is set to false. The function *plan_new_path* is then used to compute a path sequence from the current position to the goal.

A positive side effect of this termination criteria is that the segments of the original path are stored, and can be reissued as the new path sequence to the goal, beginning with the segment on the list that was involved in the intersection. It is important to remember that there may be other obstacles ahead, so the stored list of path segments must be maintained for future use. A new line must be added to the stored list: the one that began at the exterior corner of the obstacle, and intersected a path in the list. This is critical, since a future obstacle may be encountered while still following this line, and wall following termination criteria may not be met if the list of paths is not correctly updated.

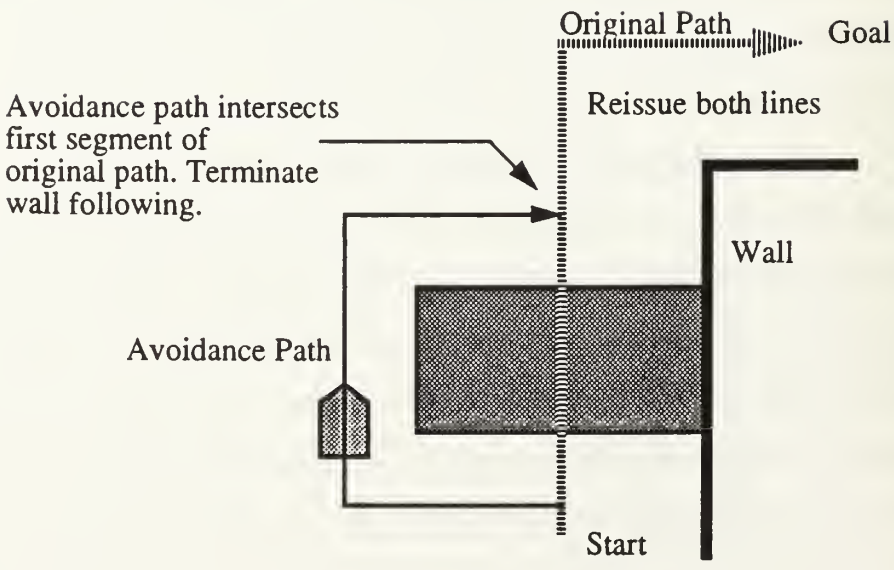
d. Recognizing a Blocked Path

In some situations, the robot may not be able to find a way around an obstacle, and the criteria for termination of wall following will never be met. There are two different cases to consider, each requires a different test in order to say that the goal cannot be reached. If the criteria are met, then the boolean *cannot_reach_goal* in Figure 3 will be set to true, and the path planner will have to resolve the situation as described above.

In the first case, the obstacle or obstacles may extend across the hallway, with no safe path around them. This situation can be identified by several tests, but the easiest one is to use the normal reversals that occur when the robot reaches an environment wall during the wall following process. There are three stages to the test. When the robot begins wall following after encountering an obstacle, the test is in the first stage. When the robot executes a reversal at an environment wall, the test moves to the second stage. If another wall reversal occurs during the same wall following sequence, the test reaches to the final stage, and the goal cannot be reached. Figure 8 shows an example of a blocked hallway. This test assumes that there is no alternate path from the start to the goal, such as by using

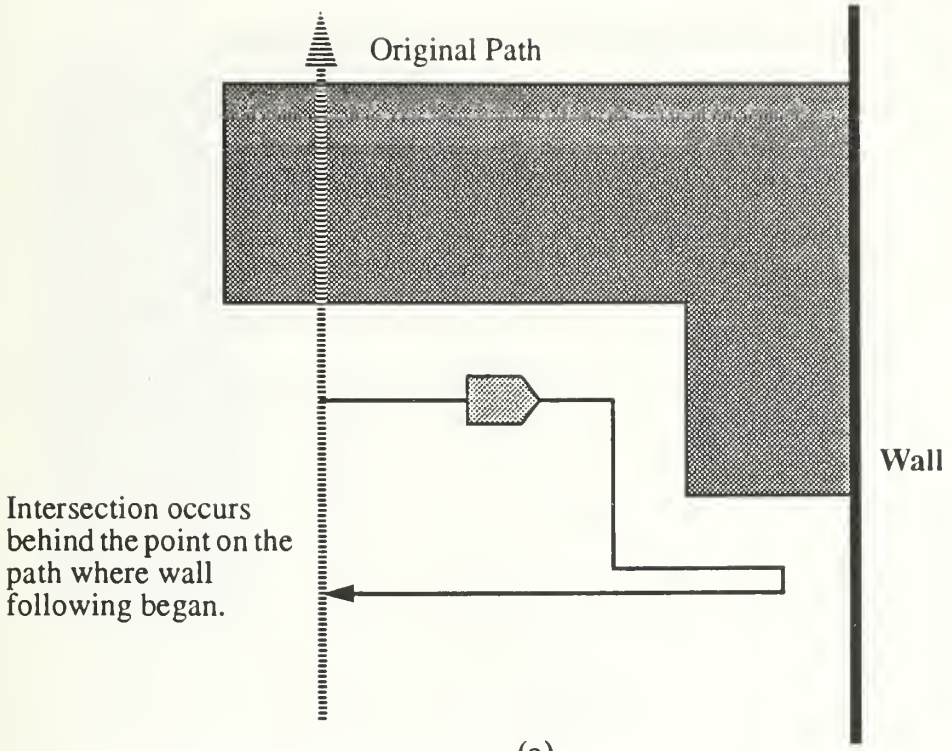


(a)

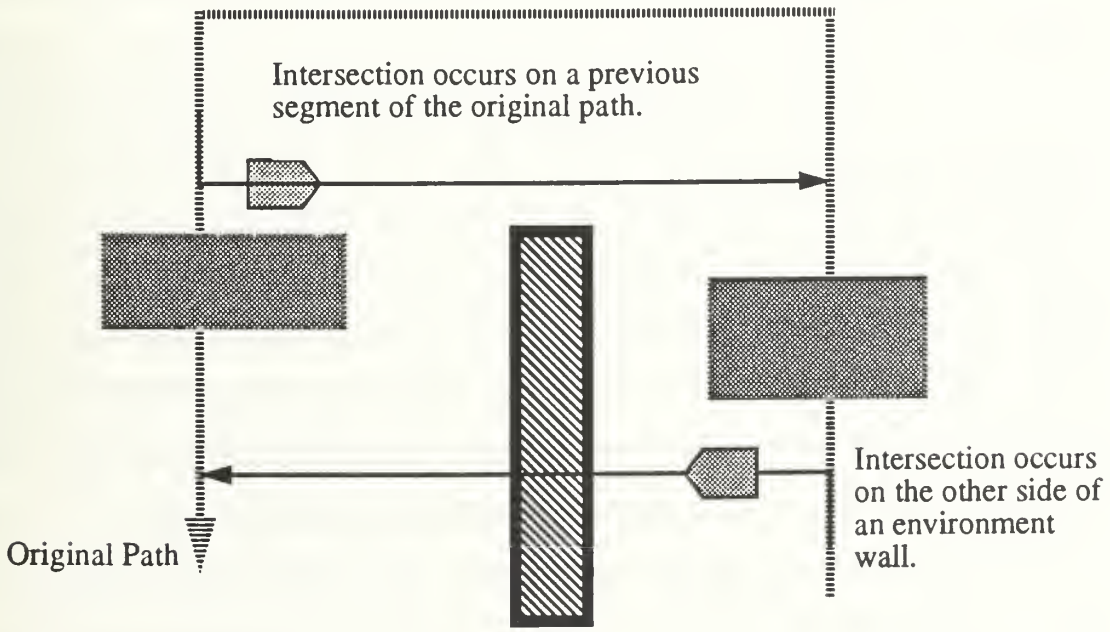


(b)

Figure 6 - Termination Criteria for Wall Following



(a)



(b)

Figure 7 - Path Intersections that Fail Termination Criteria

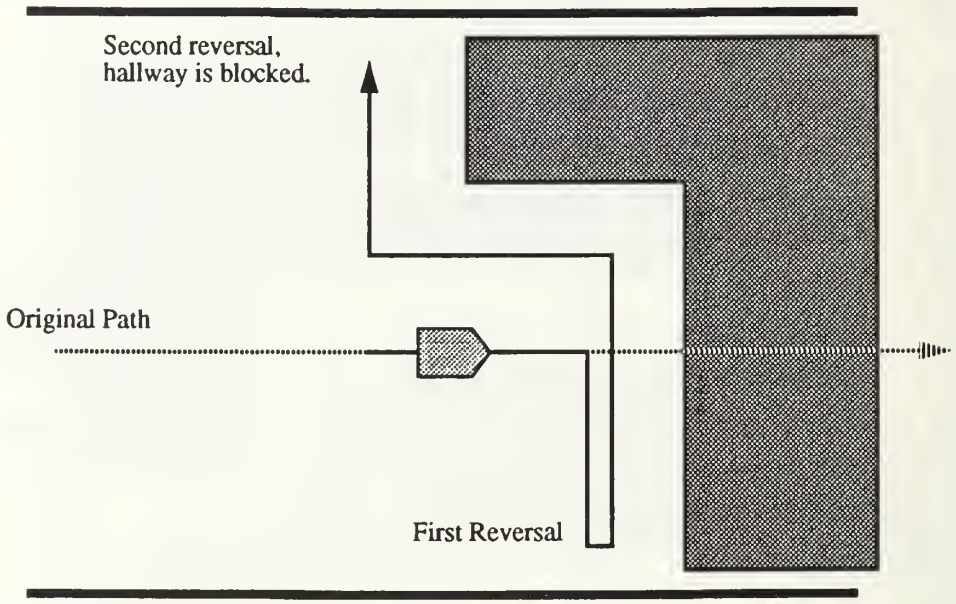


Figure 8 - Blocked Hallway

a different corridor that connects on the other side of the obstacle. Given such a corridor, the goal may still be achievable, if the path planner is sophisticated enough to use the alternate route.

The second case in which the goal is unattainable is when an obstacle is located on top of the goal position, but does not extend to the environment walls. In this situation, the three stage reversal test will not work, since the robot can circumnavigate the obstacle in either direction without encountering an environment wall. In fact, it is this ability to go all the way around the obstacle that is the key to a test that can identify the goal as unattainable. When the robot first encounters the obstacle and begins wall following, a vector from the robot to the goal is computed, and the orientation of this vector is noted. Then, as the robot wall follows around the obstacle, an incremental summation of changes in the orientation of the robot-goal vector is maintained. As the robot goes around the obstacle--and the goal, the sum will eventually reach 2π or -2π, depending on the direction of travel and the convention used for changes in vector orientation. Figure 9 shows the robot following a path S around the goal in a counter-clockwise direction; by convention

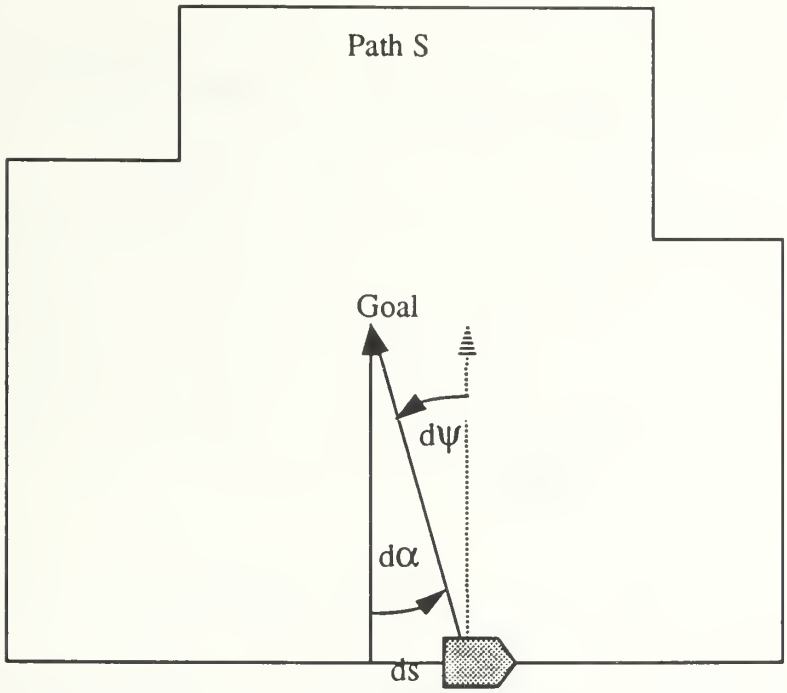


Figure 9 - Circumnavigating the Goal

this results in positive changes in ψ, the orientation of the robot-goal vector P. Let ψ_0 be the initial orientation of P, and ψ_s be the orientation of P when the robot has traveled a distance ds along the path SThen $d\psi$ is the change in the orientation of P. Note that the angle $d\alpha$ will always be congruent to $d\psi$. As the robot travels around S, the summation of $d\alpha$ must total 2π when the robot returns to the starting point, since $\Sigma\, d\alpha$ forms a complete circle around the goal. Therefore, $\Sigma\, d\psi$ must also equal 2π. For a small increment of path travel Δs, the same behavior is expected of $\Sigma\, \Delta\psi$. It is not difficult to keep a running sum of $\Delta\psi$, and when the sum reaches $\pm 2\pi$, then the boolean *cannot_reach_goal* is set to true.

If the goal position is outside the path S, then $\Sigma\, \Delta\psi$ cannot reach $\pm 2\pi$. In fact, if the robot performs a complete circuit of S, the value of the sum will be zero. As shown in fififififii the maximum value that $\Sigma\, \Delta\psi$ can attain is equal to the angel subtended by the tangents from S to the goal. Of course, if the goal is outside path S, then at some point one

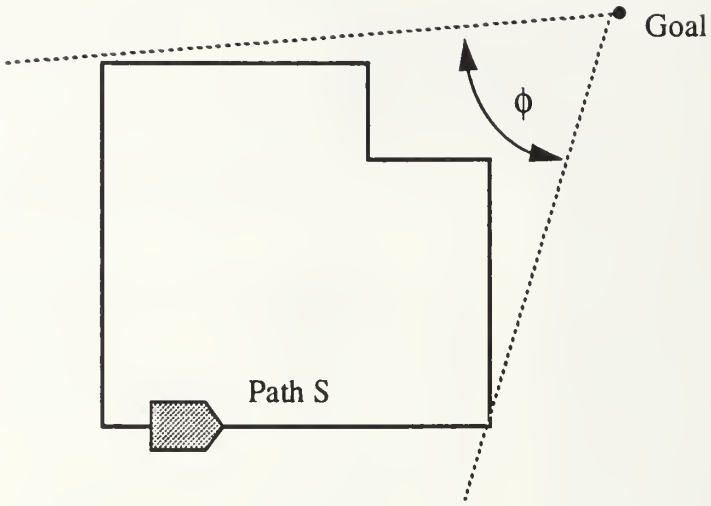


Figure 10 - Goal Position Outside the Path

of the commanded lines will intersect the original path to the goal, and wall following will terminate as described above.

While it is true that other tests may also identify this case where the goal is circumnavigated, $\Sigma \Delta\psi = \pm 2\pi$ is attractive because it does not matter how many interior or exterior corners the obstacle has. Also, the robot does not need to pass through the exact spot where wall following began, as some of the other tests may require.

e. Resolving Odometry Error

The original path sequence to the goal was computed to allow for a certain margin of error in the robot's odometry system. If the odometry error module reports that the current error estimate exceeds the normal margin, then the robot may experience problems with the environment walls even though it thinks it is following a safe pre-computed path. The first problem is that the sonar returns from the walls may be interpreted as obstacles, since the returns will not correspond with the robot's stored map of the walls when the odometry error is too large. The second problem is that the robot may actually collide with a wall, when trying to pass through a doorway, for example.

Function *resolve_odometry_error* is responsible for choosing between two possible courses of action. The best course is to perform an odometry update by driving past one of the MacPherson landmarks, normally a doorway. This will restore the odometry error to near zero, and the path to the goal can be safely resumed. [Kana93] A second course of action is to proceed without the odometry correction. In this case sonar returns from the environment walls may be interpreted as obstacles, and will probably cause extra maneuvering to avoid these 'obstacles'. If the robot must pass through a doorway, the best results will be obtained if the robot first drives by the doorway, using sonars to locate the edges, and then turns back to go through the opening. Figure 11 shows these two courses of action.

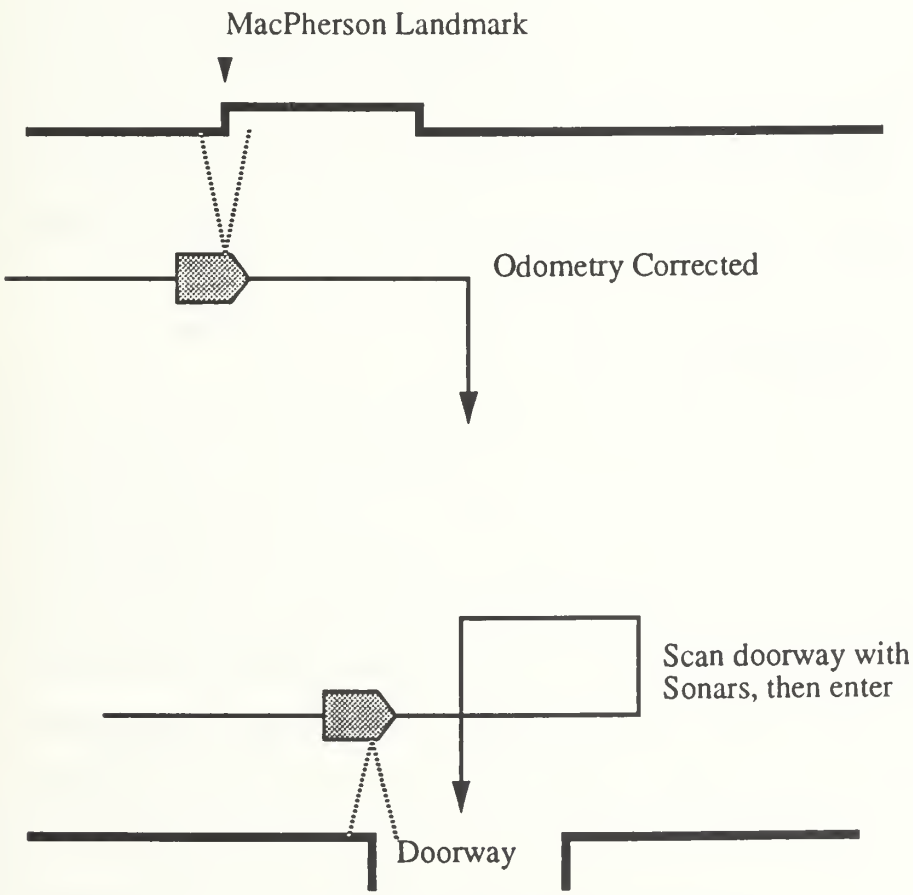


Figure 11 - Odometry Error Resolution

3. Goal Achievement Module

This subsystem is responsible for posting the position of the goal, as well as the importance rating associated with achieving that goal. The goal position and importance rating may come directly from the user, or they may be specified by a higher level control system. If there is more than one goal, such as a series of waypoints that make up a mission, then the function *get_next_goal* will take care of issuing the waypoints one at a time, in the proper sequence. The goal achievement module also updates the list of goals that have been successfully attained, using function *add_to_list*.

4. Shortest Path Module

This subsystem is responsible for computing and posting the shortest path from the current robot position to the goal. This information will be used by the path planning module when computing a path to the goal. The usefulness of this module is limited in view of the assumption that the robot will always follow orthogonal lines. Without this limitation, the shortest path module would have much more work to do in its job of assisting the path planner.

5. Wall Avoidance Module

This system is little more than a data structure, *world_map* in Figure 3, that contains the known locations of the boundaries of the environment.The module performs two functions. First, it provides wall position information to the path planning module for use when planning a path to the goal. Second, it provides wall position information to the obstacle recognition module to permit classification of sonar returns. Given the odometry position of the robot, and a sonar range, it is fairly easy to determine if the range corresponds to a wall of the environment. Spurious sonar echoes, such as from interior corners while the robot is turning, can cause false obstacles to be declared. This problem is minimized by ignoring the sonar returns while the robot is in a turn. Restricting the robot to orthogonal paths as stated in the assumptions helps prevent inaccurate ranges due to sonar beam spread.

6. Obstacle Recognition Module

An obstacle is classified as any object detected by the sensors that is not part of the known world, using data provided by the wall avoidance module. The obstacle recognition module is responsible for posting sensor information about obstacles detected. This module sets the value of the boolean *obstructing_obstacle* shown in Figure 3 to true. The information will be used by the path planner in performing obstacle avoidance maneuvers, if necessary. Some detected obstacles may not require avoidance, such as an obstacle to the side of the path. Finally, the module must pass the obstacle information to the obstacle history module, which will store the last known location of the obstacle, along with the time of detection.

7. Odometry Error Module

This subsystem tracks the motion of the robot and provides an estimate of the accuracy of the robot's current odometry position. The amount of odometry error varies with the distance traveled, the number of turns performed, robot speed, and wheel friction. Yamabico-11 can achieve as little as two centimeters of error when traveling around a ten meter long racetrack path [MacPh93]. This error estimate will be used by the path planning module when deciding how the robot is going to penetrate a relatively narrow opening, as described in the path planning module above.

8. Obstacle History Module

This subsystem is responsible for tracking the time since an obstacle was last detected by the sensors, and deciding if the object has moved. If the robot's path returns to the vicinity of the obstacle, the decision will be used to assist the path planner in choosing between a path that assumes the obstacle is still in the same place, or a path that assumes the obstacle has moved. The obstacle history data is represented by the data structure *known_obstacles* as shown in Figure 3. It is important to note that the data on an individual obstacle will probably be incomplete. The data structure will contain a sequence of line segments obtained by the sonars as the robot wall follows around an obstacle. If the robot

is able to terminate wall following after only one exterior corner, the obstacle data may consist of only one full side of the obstacle and part of the first side encountered. The ideal solution is to use a recognition process to classify the object based on the available data, and fill in the missing sides using the standard dimensions for that "object".

The study of obstacle recognition, as well as how long different obstacles remain in the same location is considered beyond the scope of this thesis. Observation shows that many of the obstacles in the corridors of Spanagel Hall are boxes set in the hallway to be thrown away. Therefore, the mean lifetime of an obstacle may be approximately one day. For the current expert system, obstacles are assumed to have short lifetimes. Therefore, information sent to the path planner will always indicate that the *known_obstacles* list is empty, unless the user specifies otherwise. This results in more emphasis on obstacle avoidance, and less emphasis on path planning. As stated in the assumptions, obstacles are assumed to be stationary with respect to the robot, so people are excluded as obstacles for this system. This assumption is convenient for another reason: the sonar sensors on Yamabico-11 do not get good returns from human legs.

IV. MODIFICATIONS TO MML

A. NEW MML FUNCTIONS

Three new functions were added to the model-based mobile robot language. These functions are necessary to be able to perform some of the tests and actions in the expert system.

1. Align

This is a simple function that takes an angle in radians, and rounds the angle to the nearest integer multiple of $\pi/2$. The normal use of the function is to assist in issuing line commands that are orthogonal to the environment. The function is implemented in C language as part of MML. The input parameter is of type double, and the aligned angle is of type double. For a given angle *input*, the value of for *align*(*input*) is:

$$\left\lfloor \frac{(input + \frac{\pi}{4})}{\frac{\pi}{2}} \right\rfloor \cdot \frac{\pi}{2}$$

2. Wall_range

This function is required to allow the expert system to analyze the position of the environment walls in relation to the robot. The normal operating environment for Yamabico-11 is the fifth floor of Spanagel Hall. The corridor has been measured and a list of x and y coordinate pairs describes the locations of each interior and exterior corner of the hallway. Function *wall_range*, using existing line segment processing routines, takes information about the robot's position, and return the range to the environment walls in any

of the orthogonal directions. Additional flexibility in the function allows the expert system to easily access environment information for two main uses.

First, by specifying one of the 12 sonar transducers, the expert system can request the range to the environment wall in the direction of that sonar axis, using the selected sonar transducer as the reference. This is used by the obstacle recognition module when classifying sonar returns as obstacles or environment walls.

Second, by specifying the robot itself, and a desired direction, the expert system can request the range to the environment wall from the center of the robot in the desired direction. This information is used to initiate a wall following reversal as the robot nears an environment wall during obstacle avoidance. The function is implemented in C language as part of the MML kernel. Its inputs are an integer to specify the robot or a sonar transducer as the reference, and a parameter of type double to indicate the desired direction when the robot's center is the reference point. Due to limitations in the existing line segment processing routines, the function will only return ranges when the desired direction or sonar axis is within 15 degrees of one of the four cardinal directions.

3. Flush

An important feature of Yamabico-11 and MML is that a series of path specifications can be sent to the robot at one time, and the robot will then follow the paths in order, performing smooth transitions between each pair of paths [Alex93]. MML uses an instruction buffer to store the pending motion commands. This capability is used by the expert system when the original path sequence to the goal is computed by the path planner. Several lines will be issued as necessary to achieve the goal. When the robot encounters an obstacle, the path planner must discard the pending motion commands and plan a new path sequence around the obstacle. The *flush* function takes the necessary steps to discard the pending instructions on the instruction buffer. The function also resets several data structures so that path transitions can be properly performed, using the line currently in use as the basis for the transition.

An important change is performed by *flush* if the current path is a bline motion command, such as when the robot is approaching the goal and preparing to stop. As described in Appendix A, the transition point from a bline to a new path is always at the endpoint of the bline. Unfortunately, the endpoint of the bline may be on the other side of a newly encountered obstacle. Even if pending instructions are discarded, MML will still use the endpoint of the bline as the transition point. The *flush* function overcomes this difficulty by converting the current bline to a line. This allows standard transition point calculations to be used, and the commands issued for wall following will be properly carried out.

During development of the *flush* function, an alternative group of functions was considered that preserved the pending motion commands. These functions would insert new motion commands in front of pending motion commands, instead of after them, recomputing transition points as necessary. One disadvantage with this approach is that significant error checking is required to prevent illegal combinations of paths. Also, the current implementation of the instruction buffer does not easily accommodate the actions required to insert a new path command in front of other commands.

B. ENHANCED FUNCTIONS

Recently MML was completely revised to implement the concept of path tracking, as well as the previous point to point navigation methods. Although the design specifications for the revised MML are nearly complete, the implementation is still in progress. One aspect that was incomplete was that the robot failed to shift to a subsequent path if it was already past the calculated transition point. This meant that even if an obstacle was detected and an avoidance command issued, if the robot was past the calculated transition point to the new line, the transition was ignored, and collision with the obstacle would occur. A partial solution involved reducing the transition distance, with a resulting sharper turn, and increasing the range at which the obstacles are classified and followed.

A permanent solution has been implemented to keep track of the robot's distance to the transition point. If the distance is getting smaller as the robot moves, then the transition point is still ahead of the robot. However, if the distance starts getting larger, then the robot is past the transition point and should immediately depart the first line and start tracking the next line. Since the robot is past the optimum transition point, there will be some overshoot before it settles down onto the second line, but the motion is still very smooth. More important, the transition occurs, and collision with the obstacle is avoided.

V. YAMABICO SIMULATOR

A. LIMITATIONS OF THE OLD SIMULATOR

The Yamabico simulator is a workstation program that models the motion and sonar systems of Yamabico-11. It uses much of the same code as the real robot, differences exist where it is necessary to simulate the robots motion and other responses to commands. The interpretation of MML commands and the calculation of path intersections and transition points is performed in the same manner as on the real robot. While the simulator accurately mimics the robot's motion and sonar returns, its biggest drawback is its sequential processing of commands, followed by execution. What this means is that the user's command program cannot use any of the MML commands that depend on robot motion, such as *wait_until*(X, GT, 100.0). This is because the simulated robot will not start moving until the entire user program has been loaded. If it does not start moving, the conditional command cannot be satisfied, and the user program cannot be completed, leading to a deadlock situation. The result is that the simulator in its current state is good for testing MML motion commands, but it cannot perform the kind of dynamic reaction and path planning that is necessary for obstacle avoidance.

B. MODIFICATIONS TO THE SIMULATOR

One answer to the single pass user's program in the simulator is to turn the simulator into a true multi-tasking system. Then the simulator would more closely model the actual robot's behavior, and allow robot motion and user commands to proceed in parallel. However, the difficulties of dealing with timing, critical sections, and inter-process communications made this approach an intimidating idea.

The solution that was employed was to break up the functions of the simulator into tasks that could be completed in one pass. Instead of loading the user commands onto the instruction buffer and then entering a loop until all the commands were completed, the

program makes a single pass through the motion control algorithm. The user's commands are still loaded onto the buffer as before, but the simulated robot will only move the distance equivalent of one time interval, normally ten milliseconds. After that motion is complete, data changes can be analyzed, such as robot position and sonar returns, and new user commands can be issued. Eventually the program makes as many passes through the motion control algorithm as before, but now the opportunity exists to respond to data changes and issue new motion commands as required.

In support of this approach, the simulator was divided into three main parts: robot initialization, graphics initialization, and robot motion. The first two parts are executed once, at start up, while the robot motion functions are executed thousands of times. To avoid repeatedly passing parameters, the arguments to the robot motion section were converted into global variables.

An additional modification to the simulator was required to simulate obstacles. The original simulator displayed only the hallway that Yamabico-11 normally operates in. The data storage and processing routines were modified to allow two obstacles to be displayed, wherever the user desires. Other functions were changed to ensure that the two obstacles were included when computing simulated sonar returns.

C. IMPLEMENTING THE EXPERT SYSTEM IN CLIPS

1. Grid Based Expert System

A key stage in the evolution of the expert system for Yamabico-11 was the development of an expert system robot simulator that assumes a grid layout for the environment. The size of each grid square is approximately the same size as the robot. The robot moves in increments of one grid square, and can sense objects only in the adjacent grid squares. The size of any obstacle is a multiple of one grid square, and obstacles are placed so that their edges align with the edges of one or more grid squares.

The grid based expert system is implemented entirely in CLIPS. The underlying algorithm for goal achievement is to always move in the direction of the goal. When

obstacles are encountered, the robot shifts into a wall-following mode, and follows the edge of the obstacle until the termination criteria are met. While wall-following around an obstacle, the robot may temporarily move away from the goal. Various criteria for termination of wall-following were tested. Several simple tests worked well for most situations, but the robot occasionally failed to achieve the goal when faced with a certain obstacle arrangement. The best criteria involves measuring the distance to the goal. While wall-following, when the robot reaches a grid square that is closer to the goal than the square where wall-following started, then the robot terminates wall-following. This criteria can lead to frequent initiation and termination of wall following, often terminating after moving only one square. However, the results justify the criteria: the robot always achieved the goal, if the goal was attainable.

The grid based expert system is not directly applicable to Yamabico-11 because Yamabico-11 moves in a much more continuous manner, rather than from one grid square to the next. The grid based expert system provided the basis for many of the rules in the final expert system. Changes were made to allow for the robot's continuous motion and the limitations of the sonar sensors. The distance to goal criteria for termination of wall following is not suitable for Yamabico-11, due to the frequent starting and stopping of wall following. A complete discussion of the grid based expert system, and the CLIPS code, is included as Appendix B.

2. Interfacing CLIPS with the Simulator

The reason for implementing the expert system in CLIPS is to take advantage of the easy translation of behavior rules into CLIPS rules, while still providing capability for easy interface with the simulator functions, and possibly the actual Yamabico-11 robot. The goal was to minimize the need for interaction between the expert system and the simulator functions, by treating the expert system as an independent module, getting data from the simulated robot, and passing commands back to the robot based on the rules of the expert system.

CLIPS can be modified to interact with other high-level languages in any of three basic methods. In this case, the interacting language is C, since the existing Yamabico simulator is programmed in C. First, CLIPS can be used as the top-level program, using external function calls to access the simulator. Second, a CLIPS program can be called from inside the simulator, excuting the CLIPS program and then retruning to the host. In both of these methods, the CLIPS programs can be modified without re-compiling the simulator's source code. Finally, a CLIPS program unit can be compiled and then linked with the simulator's object code, creating a single run-time module. [Giar91], [NASA91] For the puposes of implementing the expert system on the simulator, the first method was selected, using CLIPS as the top-level program. This was done to overcome the single-pass limitation of the original simulator in the same mocules that contained the expert system, so that the interactions were clear.

The resulting interface uses ten functions to allow CLIPS to interact with the simulator. Two of the functions, *init_sim* and *run_sim* are for operation of the simulator, three functions receive data from the robot, and five functions pass MML commands to the robot. A brief description of each function follows.

-- *init_sim* Starts the simulator by calling the simulator functions that initialize the variables, and sets up the graphics display.

-- *run_sim* Directs the simulator to advance the robot the appropriate distance equivalent to one time interval, normally ten milliseconds.

-- *get_vehicle* Reads the current robot's position and orientation and returns the data to the expert system.

-- *get_sonar* Reads the current robot's sonar data for the left, front, and right sonars, and returns the data to the expert system for analysis.

-- *wall_range* This function is used by the obstacle recognition module and the path planner to read the distance from the robot to the walls of the environment, using the stored map of the environment.

-- *clips_set_rob, clips_rotate, clips_line*, and *clips_bline* These functions are used to issue the corresponding MML motion commands to the robot.

-- *clips_flush* Discards pending motion commands on the robot's instruction buffer. Used when avoiding obstacles.

The function *wall_range* could have been implemented in CLIPS, reducing the number of interface functions needed. However, convenience dictated that they be implemented in C, and included as part of the simulator. The simulator already contained the data structures and segment handling routines necessary for these functions, and the functions would have to be translated to C later, to implement the expert system on Yamabico-11.

3. Implementing the Expert System in CLIPS

The CLIPS rules that make up the expert system on the modified simulator have two separate tasks. The first task is to drive the modified simulator, by calling the initialization functions, and then repeatedly calling the motion control algorithm. The second task is to implement the behavior rules of the expert system in order to issue the correct commands to the robot. At the time the simulator was modified, it was clear that CLIPS could not be used on the actual Yamabico-11, as discussed in Chapter VI. Therefore, once the CLIPS rules were developed and tested on the simulator, they would have to be translated into C for implementation on the actual robot. This knowledge affected the style of the CLIPS programming slightly. For example, more CLIPS functions were used than might have been used otherwise, since the functions are easily converted to C. Also, the CLIPS program was designed to operate in a somewhat sequential manner, by using salience and control facts.

The CLIPS rules were organized by the modules they supported, and also by their purpose. Several rules were used simply to drive the simulator and return robot data to the expert system. These rules used salience and control facts to form an endless loop. Inside

this loop, other rules examined the latest data from the robot and the world, and issued new commands based on the behavior rules of the expert system.

Not all of the expert system was implemented on the simulator. For example, the odometry error module was left out. To perform an odometry correction, the nearest MacPherson landmark would be issued as a drive-by goal, and given priority over the current goal. Once the odometry correction is complete, the desired goal can be re-issued. compared to obstacle avoidance. The CLIPS programs and interface functions are included as Appendix C.

D. RESULTS WITH MODIFIED SIMULATOR

The results of testing the expert system on the simulator were extremely valuable. The knowledge gained supported the decision to implement the expert system first in CLIPS on the simulator, and then in C on Yamabico-11. The simulator provided a focused system, free from multi-tasking and sonar problems. Changes can be quickly made and tested, with repeatable results; features that Yamabico-11 cannot always deliver.

Frequently, when developing CLIPS rules to implement a portion of the expert system, strange results were observed. Using the CLIPS debugging commands, the fact database and the rule firings could be traced one rule at a time, and the reason for the unexpected behavior could be isolated. If necessary, the run could be repeated, with the assurance that the same behavior would occur, except for changes due to round off error. Troubleshooting the same problem on Yamabico-11 would be very difficult, due to the multi-level interrupts, and poor I/O and debugging support.

Testing obstacle avoidance routines on the simulator played a key part in the success of the expert system on Yamabico-11. Various criteria for termination of wall following were tried, until the test for intersection with the original path proved best. As described in Chapter III, other criteria exist, but these gave awkward and jerky results when tested on the simulator. Various constants, such as how far away from an obstacle should the robot

be while wall following, were implemented, tested and revised using the modified simulator.

Another area where the simulator proved its worth is the performance of turns. When executing a turn in front of an obstacle, it is necessary to complete the turn before making any further decisions. This is because the sonars are modeled with a thirty degree beamwidth in the simulator. During the turn, the sonars will not get returns to the obstacle, and it will appear that the obstacle is no longer there. If a path decision is made during this period, a collision could occur. The rule that handled this situation forced the path planner to wait until the robot was nearly orthogonal after the turn before making any decisions.

A sample run of the modified Yamabico simulator is shown in Figure 12.

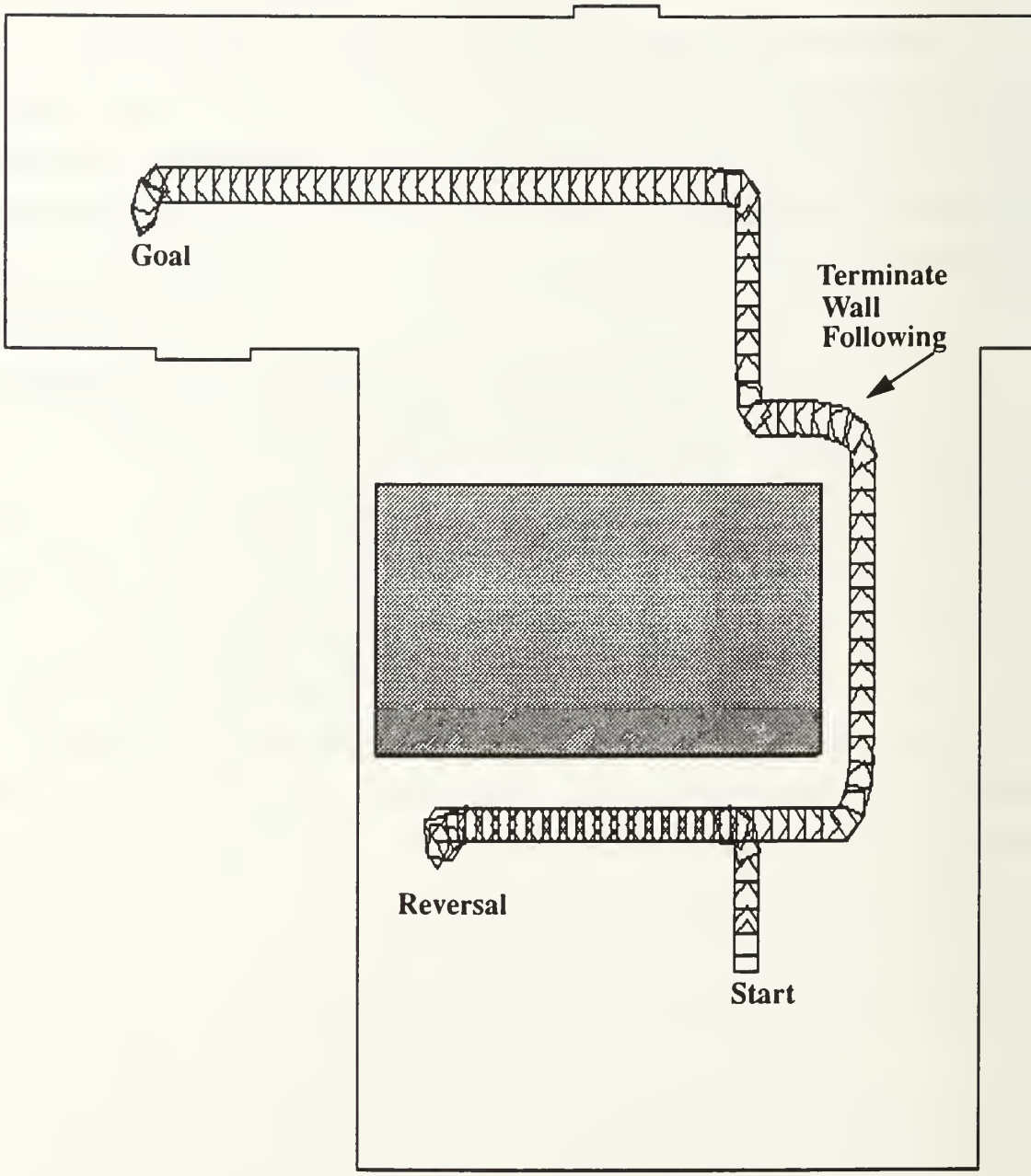


Figure 12 - Sample Run of Modified Simulator

VI. THE EXPERT SYSTEM ON YAMABICO

A. EARLY ATTEMPTS TO EMBED CLIPS

As described in Chapter V there are three methods for combining CLIPS with other languages. Unfortunately, none of these methods worked on Yamabico-11. Although the modified simulator used CLIPS as the top-level program, the best approach for Yamabico-11 was to implement the expert system as a CLIPS module. This module would be repeatedly called by the user's program on Yamabico-11, whenever a decision regarding the correct path to follow arose. In this way, the expert system would act as a consultant, planning and issuing the path sequence to the goal, and then becoming inactive until the robot reports the goal has been achieved, or a new path is needed, to avoid an obstacle, for example.

Unlike the simulator, the source code for Yamabico-11 must be compiled on a SUN 3/60 workstation and downloaded onto the robot. There are no standard libraries available at compile time, since these routines are not implemented on the robot. All input/output and supporting math functions are home made versions that allow the robot to operate without the UNIX operating system, using only the BUG monitor resident on the processor.

The reasons that CLIPS could not be interfaced with Yamabico-11 are not clear. The symptoms of the problem are that the CLIPS modules will not link with the Yamabico-11 source code due to undefined functions. The functions that are undefined are those standard input/output and math functions that the CLIPS code expects to be visible, but that are purposely not included at compile time due to the lack of library support on the robot.

Even when the CLIPS routine was compiled separately into a run-time unit, and then linked with the Yamabico code, using the third method described in Chapter V, undefined functions prevented successful linking. Finally, attempts were made to fool the linker, by creating a file with dummy functions that had the same names as the undefined functions.

With this dummy file available, the linker still failed to link, citing additional undefined function names that could not be found anywhere in the CLIPS source code. [Giar91]

B. IMPLEMENTING THE EXPERT SYSTEM IN C.

Although CLIPS could not be used with the Yamabico-11 source code, the expert system could still be implemented on the robot. The implementation consisted of converting the CLIPS rules and functions into C source code, by hand, and compiling the expert system along with either the MML kernel, the user's program, or both. Ideally, the entire operation of the expert system will be hidden from the user, so that he only calls the expert system as a function call to *execute_mission*, as shown in Figure 3. For development purposes, about half of the supporting functions for the expert system were left in the user program, since that module can be modified, compiled and downloaded much faster than the kernel [User93]. As the functions were tested and found to be reliable, they were moved to the kernel to keep the size of the user module small.

The time spent implementing the expert system in CLIPS on a workstation simulator proved extremely valuable when it came to programming the expert system in C. Most of the CLIPS rules and functions were easily translated into C. Two structures were created, to hold data about obstacles and the robot status. The first structure, OBSTACLE_TYPE, contains three boolean fields, for ahead, left, and right, that are set to true when an obstacle is within the range specified by a defined constant. The second structure, ROBOT_TYPE, contains three boolean fields. Two fields are used to indicate that the robot is wall following on the right or left side. The third field is used to keep track of the three stage reversal test that indicates a blocked hallway. These structures were useful because they could be used in a manner similar to the facts database in CLIPS.

Some of the CLIPS rules and functions were not needed for Yamabico-11. For example, the three rules that operated the locomotion portion of the simulator were extraneous. Also, the rules that watched for certain values of the robot's x, y, or theta coordinates were unnecessary. Consider a right turn at an interior corner, where a trip value

is set for the robot's theta coordinate. In the CLIPS simulator, the main loop had to keep cycling while the decision making rules were held back. However, Yamabico-11, with its multi-level processes, can afford to busy wait in the foreground, while the higher interrupt level are busy steering through the turn.

The outer procedure call, *execute_mission*, as shown in Figure 3, was not implemented. Instead, a series of configurations were defined in the user's program, representing the robot's start and goal positions. For the present implementation only one goal can be specified at a time. This allows the elimination of the outer loop of the algorithm, along with the decisions relating to going on to the next goal if the current one is unattainable. Then the function plan path to goal is called, which calculates and issues the two or three lines needed to get to the goal. Finally, the robot enters a loop, where it follows the commanded path, while watching for obstacles. If an obstacle is detected, avoidance paths are computed and issued, as described in Chapter III, until the obstacle is no longer obstructing. At that point, the remainder of the original path is reissued, the robot continues on to the goal, watching for more obstacles.

The ability to perform reversals off the environment walls has not yet been implemented. The routines that were used in the simulator for the polygon simulations of the world are not available to the robot, and other routines will have to be substituted.

As in the simulator, the odometry error module was not implemented. The logical complexity of the many if-then-else statements required to convert the CLIPS rules to C made it too difficult to consider the case of excessive odometry error.

C. RESULTS OF TESTING ON YAMABICO-11

Early results with Yamabico-11 were promising. The path planning functions and obstacle detection routines worked well. Problems were encountered when the wall following ranges and other constants that had been refined in the simulator did not work on Yamabico-11. The reason was that the robot was detecting the obstacle and correctly issuing avoidance paths, but the robot would not turn onto the new line. This led to the

rework of the transition point test as discussed in Chapter IV, so that the robot would turn onto the new line, even if it is already past the transition point. Once that modification was completed, the robot transitions were very smooth, with close tolerances on wall following distances, even around corners.

An interesting difference between the simulator and Yamabico-11 is the constant used for delaying decisions after interior turn. In the simulator, the constant used was equivalent to 12.6 degrees. That is, decisions were blocked until the robot's theta coordinate was within 12.6 degrees of the new heading. On Yamabico-11,the highest reliable value for the constant was 10.3 degrees. This is probably due to an infrequent miss on the part of the side-scanning sonars. The side sonars do not perform as well as the upgraded forward ones, and all it takes is one missed return to signal the detection of an exterior corner. One possible solution to this is to require two missed returns in a row before an exterior corner is declared.

Finally, as indicated by other members of the Yamabico research group, there appears to be a noticeable difference in sonar quality between the wooden particle board practice obstacles and cardboard boxes or the walls. The particle boards give generally worse results than the other targets.

Many test runs were made, from different directions, and with different obstacle arrangements. It is possible to place the obstacles so as to take advantage of the delay in decision making while turning, causing a large overshoot while executing the transition to a new line. In some cases, the overshoot led to a brush with the obstacle while turning away. This problem can be solved by reducing the size constant in the transition point calculations, causing very sharp turns with very little overshoot. A disadvantage to this solution is that the robot's motion is no longer smooth, unless the speed is also reduced.

The experiments performed demonstrate that the theory of the expert system is sound. The behavior rules that "an expert" uses have been successfully implemented both on the simulator and on Yamabico-11. While further testing is still recommended, the expert system will perform its function of getting the robot to the goal without collision.

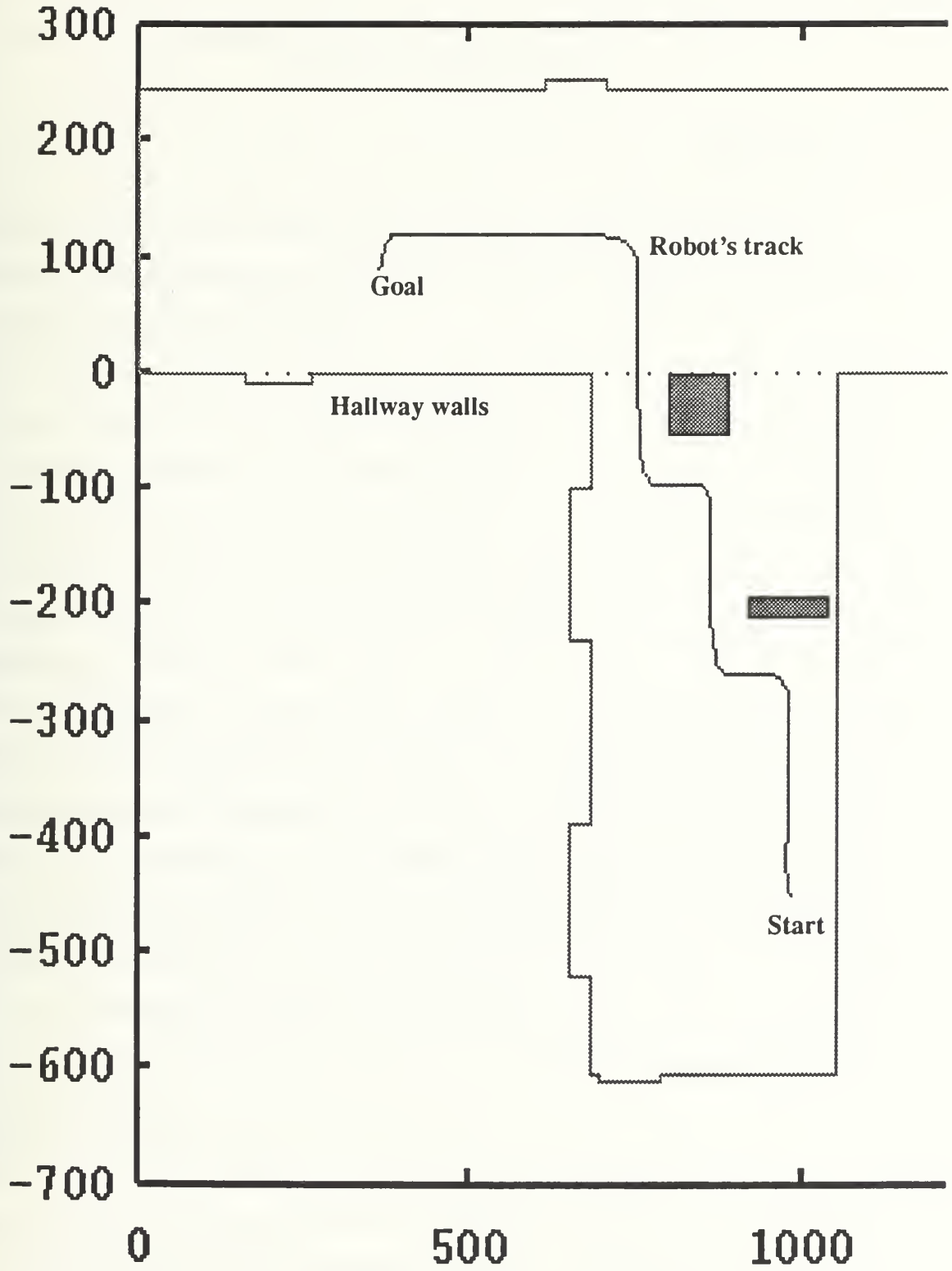


Figure 13 - Sample Yamabico Avoidance Plot

VII. CONCLUSION

A. SUMMARY

The use of expert systems to control the actions of trainers, simulators, and robots is an excellent demonstration of the capability and flexibility of a well designed expert system shell. CLIPS expert systems have been recently implemented on graphics simulators to control simulated ships and aircraft, releasing humans to use the simulators for higher levels of training [Schmidt93]. The area of obstacle avoidance, with the numerous possible arrangements of obstacles, is well-suited to the application of expert system technology.

Autonomous mobile robots have been successfully used to deliver mail, supplies, and food in office buildings and hospitals, demonstrating both the need and the ability to use mobile robots in repetitive, multi-goal tasks. However, many of these robots are constrained to following painted stripes or magnetic strips in the floor, and obstacle avoidance may mean stopping until the obstacle gets out of the way. Yamabico-11 is truly autonomous; it does not need any stripes or guides to navigate. With the addition of reliable obstacle avoidance behavior, Yamabico-11 is well on the way toward a new level of motion control. This expert system is a big step in the right direction. Although the behavior rules have been developed to anticipate all the possible obstacle arrangements permitted by the assumptions, extensive testing is still needed to validate the system, and additional behavior rules may permit some of the simplifying assumptions to be eliminated.

B. FUTURE POSSIBILITIES

There is a great deal of work still to be done in the area of obstacle avoidance and high level motion control on Yamabico-11. It is felt that the expert system approach is the only way that complex and arbitrary behavior rules can be easily applied to an autonomous mobile robot such as Yamabico-11. All that is needed is a person to ask two questions.

First, how would *I* react if faced with that situation? Second, how can the situation be recognized, and the appropriate reaction specified at the robot's level?

Ongoing work on Yamabico-11 includes the replacement of the SUN 3 onboard computer with a SPARC processor. The improved capabilities of this processor, along with the larger library provided, may make it possible to successfully embed compiled CLIPS program modules in the user program. This would enable the development of a more complex rule-based expert system in CLIPS, without the need to translate the rules to C before downloading onto the robot.

Additional work in vision and image understanding should enhance the ability of the robot to examine obstacles and plan avoidance paths. While not yet ready for real-time operation, experiments show that the coordinates of the edges of an obstacle can be computed using an image and one sonar range. This would eliminate the need for much of the complex wall following that was used in the expert system. The rules for using this image information can easily be added to the expert system, greatly increasing its capability.

The possibility of determining if an obstacle is moving is an area that needs research. Given successive ranges (assumed to be from the same object) apparent motion toward or away the robot can be calculated. By subtracting the speed of the robot (for an object ahead of the robot) from the apparent speed of the obstacle, it can be determined whether or not the obstacle is moving toward or away from the robot. However, it is much more difficult to determine lateral speed of an object, given the limitations of the sonar system. If actual object speed and direction can be calculated, then the relative motion of the object and the robot can be determined, and a new path and/or speed command can be issued to the robot to avoid collision.

APPENDIX A

SUMMARY OF MML MOTION COMMANDS

1 Define a Robot Configuration Variable (def_configuration)

Syntax: CONFIGURATION def_configuration(x, y, t, k, &p)

```
double x;
double y;
double t;
double k;
CONFIGURATION p;
```

Description:

Assigns the four parameters necessary to specify a configuration. The parameters x and y define the vehicle's location on the cartesian plane. The parameter t represents the vehicle's orientation and k represents k, the curvature of the vehicle's current motion.

Function Call: def_configuration(x, y, t, k, &p);

2 Set Robot Configuration Sequential (set_rob)

Syntax: void set_rob(q)

```
CONFIGURATION q;
```

Description:

Sets the robot's odometry configuration in a sequential manner. This function is used normally at the start of the MML program to tell the robot where it is initially. Subsequent odometry resets are also made using this function.

Function Call: set_rob(&q)

3 Move While Tracking a Line (line)

Syntax: void line(q)

```
CONFIGURATION q;
```

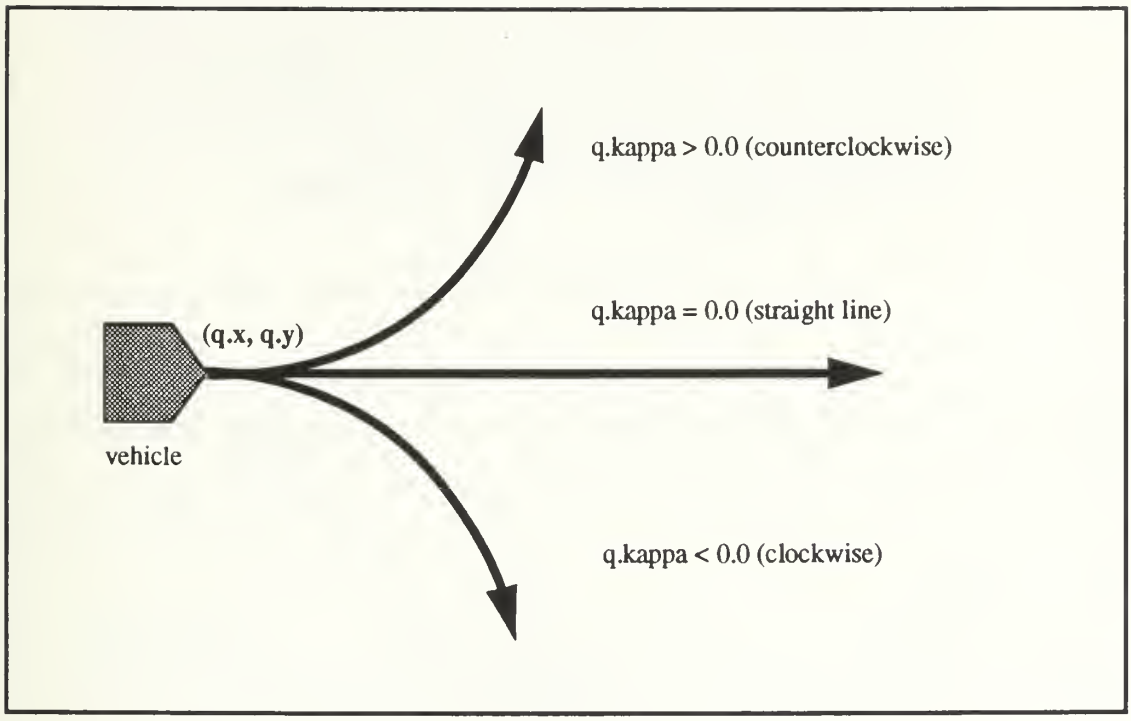


Figure 1 - The Line Function

Description:

Command that orders the robot to follow the line specified by the configuration *q*. If the path curvature is zero then *q.kappa* = 0.0. This means that the path represents a straight line passing through the point (*q.x*, *q.y*) with orientation *q.theta*. If the path curvature is nonzero, then the robot follows a circular path. When the value of k is less than zero then the vehicle's direction of motion on the circle is clockwise, and when k is greater than zero, then the motion is counterclockwise. These concepts are illustrated in Figure 1. Speed is automatically reduced to allow the robot to make sharp turns. This is reflected by the dependency between k and the vehicle speed. In simple terms, the vehicle speed must be reduced to allow it to move safely with larger values of k.

Function Call: line(&q);

4 Move While Tracking a Backward Half Line (bline)

Syntax: void bline(q)

CONFIGURATION q;

Description:

Follow the backward line specified by a configuration *q*. Upon reaching config-

uration q, transition to the next motion command. The robot should stop at the configuration specified, if the bline is the last motion command. See Figure 2 for an illustration. In case 1, the vehicle image falls on the half-line and the robot tracks as in the line function. In case 2, the vehicle's image does not fall on the half-line, the robot should transition immediately, or, if the bline is the last motion command, the robot should stop as soon as possible.

Function Call: bline(&q);

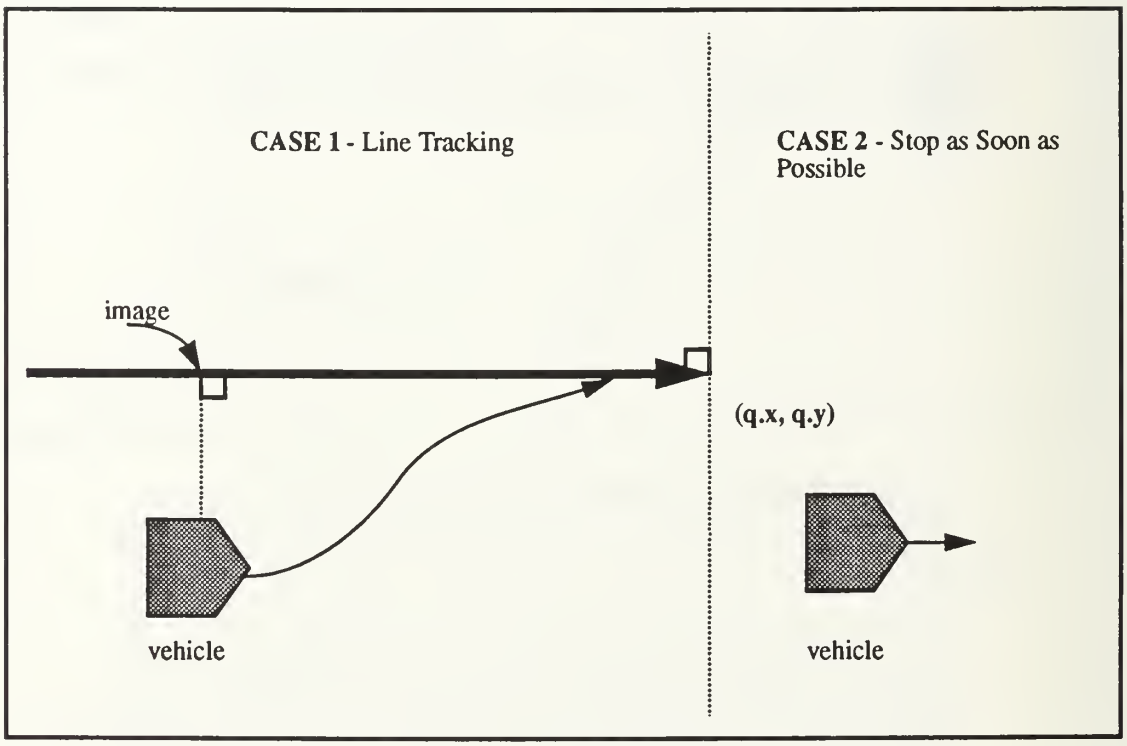


Figure 2 - Backward Line Tracking

5 Rotate the Robot in Place (rotate)

Syntax: void rotate(value)
double value;

Description:

This function causes the robot to rotate in place by the specified amount value expressed in radians. Following the convention, a positive value means rotation in the counterclockwise direction. A negative value is an order to rotate clockwise. The robot must be in a stopped condition first in order to rotate, otherwise an error occurs. For instance, the robot can rotate at the end point of a bline instruction.

Function Call: rotate(value);

APPENDIX B

GRID BASED EXPERT SYSTEM

SUBMITTED FOR CS4311

6 Summary

The expert system will find a path, if one exists, from a start position to a goal position in any environment, avoiding collisions with obstacles and boundary walls. Certain simplifying assumptions were made.

7 Assumptions

The environment consists of a rectangular room, although more complex shapes can be created by adding obstacles. The room is divided into grid squares, with unit dimensions of convenient size. For the Yamabico robot, a good unit size is 100 centimeters by 100 centimeters. Obstacles are assumed to be rectangular, and can be arranged in any fashion subject to the following constraints:

-- The size of the obstacle is a multiple of one grid square.

-- Obstacles are placed so that their edges align with the edges of one or more grid squares. See examples attached.

-- The robot moves one grid square at a time, to any of the four squares that share edges with the current location. The robot's sonar will always detect the presence of an obstacle or boundary wall in one of the adjacent grid squares.

8 Implementation

The expert system is programmed entirely in CLIPS, and is broken down into five modules. These are described below.

Intro:

This module introduces the system and allows the user to select one of the five different obstacle arrangements shown in Appendix 1. Other arrangements or room sizes are easily accommodated by changing the associated facts in the data module.

Data:

This module defines the room dimensions, the obstacle locations, and the robot start and goal positions in x, y coordinate fashion. An obstacle that is larger than one grid square is defined as separate obstacles on each of the grid squares occupied.

Setup:

This module initializes the robots parameters to the start position. It also contains rules that detect the achievement of the goal, print the robots current position as it moves around the room, update the path history, and display the path taken upon completion.

Basic Movement:

This module contains rules that control the robots movement when the next square in the direction of the goal is unobstructed. The robot tends to seek the goal in the y direction first, then move towards the goal in the x direction. This is due to the default agenda priority scheme in CLIPS, which is similar to a stack. This module also contains rules to detect the presence or absence of obstacles and boundary walls in the adjacent grid squares.

Wall Following:

This module contains rules that control robot motion when there is an obstacle adjacent to the robot in the direction that it needs to move. The rules shift the robot into a wall following mode, where it will follow the wall until it reaches a square that is closer to the goal than when it started wall following. Other rules detect when the direction of wall

following should be reversed, and when the goal cannot be achieved because it is blocked by obstacles.

9 Expert Heuristics

Unobstructed Motion:

This heuristic, though simple, directs that the robot should move towards the goal when possible, if not in a wall following mode.

Wall Following:

There are several heuristics associated with wall following:

-- When blocked by an obstacle, commence wall following with the wall on the right if the goal is to the left of the path that would be taken in unobstructed motion. Otherwise, commence left wall following. In other words, go around the obstacle in the direction of the goal when starting wall following, unless the goal is directly ahead. In this case the robot arbitrarily turns right.

-- If a boundary wall is reached during wall following, turn 180 degrees, and wall follow with the wall on the other side.

-- Continue wall following until a square is reached that is closer to the goal than the square where wall following began.

-- If two boundaries are reached during the same wall following sequence, each requiring reversing the direction of wall following, then the goal cannot be reached. The square where wall following last started is the closest (or tied for closest) that the robot can get to the goal.

10 Observations

Salience:

The use of salience to set rule priority has been minimized. Salience is used to force robot actions to be as natural as possible. For example, the robot checks for obstacles or boundary walls in adjacent squares before deciding which way to move.

Upon entering a new square, the robot always performs the following series of actions; the order of these is determined by rule salience.

-- Print the current position, and add the current square to the path history.

-- Check to see if the goal has been reached, if not, which direction is the goal from the current position.

-- Check to see if obstacles or boundary walls are in the adjacent squares.

-- If wall following, check to see if wall following should be terminated, reversed, or that the goal cannot be reached.

All other rules are scheduled according to their pattern matches and the default agenda scheduling scheme.

Wall Following Termination

Initially, the criterion for terminating the current sequence of wall following was simpler. If during the process of following the wall, the robots heading returned to the desired heading at the square where wall following started, then wall following should end.

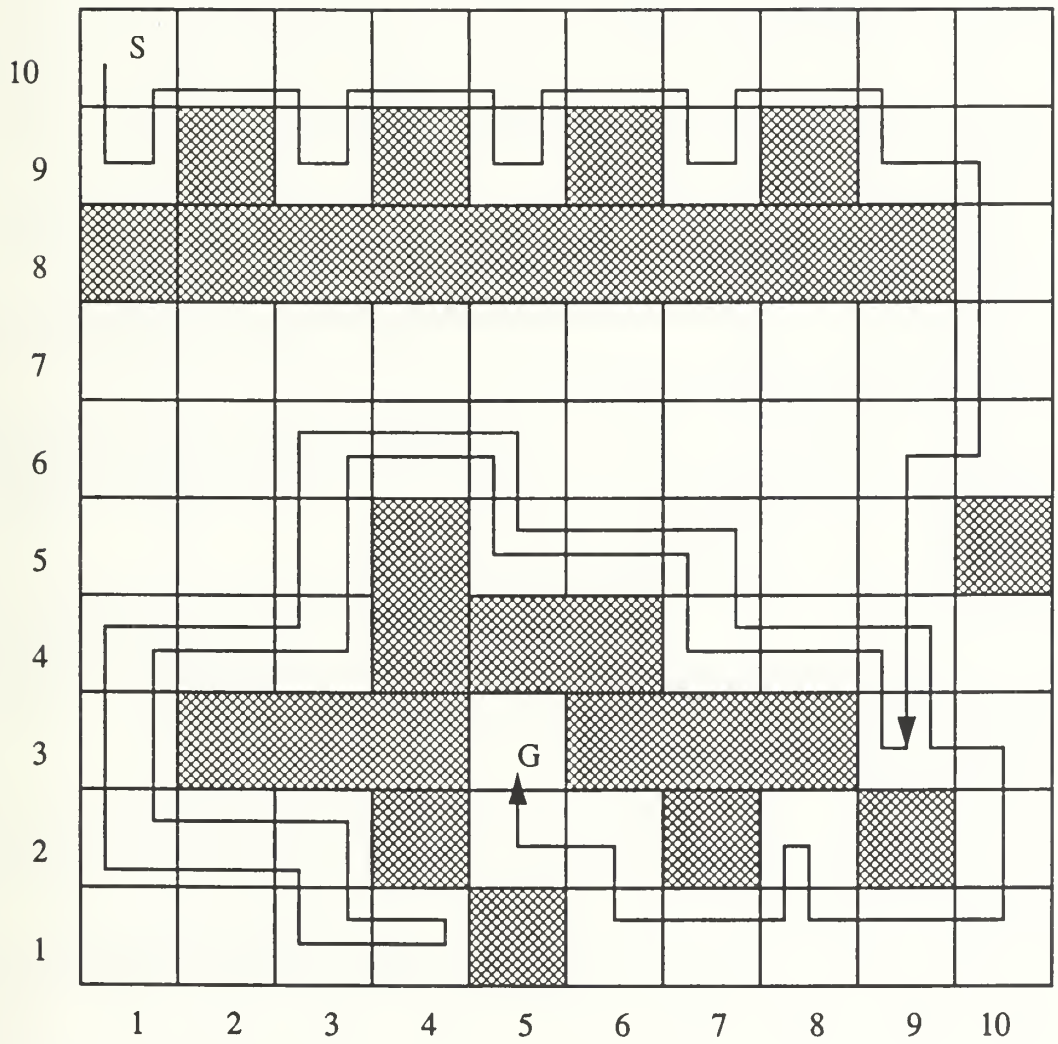
While this heuristic was good enough to get the robot around simple obstacles, and some complicated obstacles, it failed on more complex obstacles. Also, it required additional testing to prevent initiating wall following in the same direction from the same square.

11 Attachment 1-- Sample Obstacle Arrangement

A sample obstacle arrangement is shown, with start and goal positions and the path that the robot takes in its attempt to find the goal.

ATTACHMENT 1

Crenellations One



APPENDIX C

THE EXPERT SYSTEM IN CLIPS

```
;; ****************************************
;; Filename:   clips.init
;; Purpose:    Contains initialization operations
;; Author:     Bob Fish
;; ****************************************

;; Define global constants, equivalent to #define in C.
(defglobal
   ?*theta-range*      = 0.780
   ?*Beamwidth*        = .22
   ?*corner-offset*    = 50.0
   ?*corner-line-dist* = 60.0
   ?*obst-offset*      = 30.0
   ?*obst-range*       = 60.0
   ?*REVERSAL-RANGE*   = 75.0
   ?*UTURN-DELAY*      = 30.0
   ?*PI*               = (pi)
   ?*DPI*              = (* 2.0 (pi))
   ?*HPI*              = (/ (pi) 2.0)
   ?*-HPI*             = (- 0.0 ?*HPI*)
)


;; Define start and goal position
(deffacts data
   (start-posit 800.0 -450.0 ?*HPI*)
   (goal-posit  150.0 120.0 ?*PI*)
)
```

```
;;*****************************************
;;Filename:   clips.funcs
;;Purpose:    Defines functions for use in many  rules
;;  Author:     Bob Fish
;;*****************************************

;; Normalize an angle to range -PI < angle <= PI
(deffunction norm (?theta)
   (if (>= ?theta 0.0) then
    (while (> ?theta ?*PI*)
     do
     (bind ?theta (- ?theta ?*DPI*)))
    else
     (while (<= ?theta (* -1.0 ?*PI*) )
      do
      (bind ?theta (+ ?theta ?*DPI*)))
    )
    ;; return the value of ?theta
    ?theta)

;;-------------------------------------------------
;; Right and left turn functions, takes orientation, and
;; returns aligned angle + or - HPI

(deffunction right-turn-func (?orig ?delta)
   (bind ?new (- (clips_align ?orig) ?delta))
    ;; return the value of ?new
    ?new
)

(deffunction left-turn-func (?orig ?delta)
   (bind ?new (+ (clips_align ?orig) ?delta))
   ;; return the value of ?new
   ?new
)

;;-----------------------------------------
;; Four functions to determine which cardinal direction the robot
;; is facing.

(deffunction northp (?heading)
  (if (< (abs (norm (- ?heading ?*HPI*))) ?*theta-range*) then
    TRUE
   else
    FALSE
  )
)
```

```
(deffunction southp (?heading)
  (if (< (abs (norm (- ?heading 4.7123))) ?*theta-range*) then
    TRUE
   else
    FALSE
  )
)
(deffunction eastp (?heading)
  (if (< (abs (norm ?heading)) ?*theta-range*) then
    TRUE
   else
    FALSE
  )
)
(deffunction westp (?heading)
  (if (< (abs (norm (- ?heading ?*PI*))) ?*theta-range*) then
    TRUE
   else
    FALSE
  )
)

;;-----------------------------------------------
;; Function to compute the square of the distance to the goal
(deffunction sqdist (?xg ?yg ?xc ?yc)
     (bind ?xdiff (- ?xc ?xg))
     (bind ?ydiff (- ?yc ?yg))
     (+ (* ?xdiff ?xdiff) (* ?ydiff ?ydiff)))


;;-----------------------------------------------
;; The main workhorse of function plan_path_to_goal.
;; Takes start and goal positions and figures out two or three
;; lines to achieve goal.

(deffunction originate-lines (?xs ?ys ?ts ?xg ?yg ?tg)

  (if (> ?ys 20.0) then
    (if (< ?yg 20.0) then
      ;; start in hall, end in elevator, line, line, bline
      (if (< ?xs 850.0) then
        ;;first line theta = 0.0
        (assert (line 1 ?xs ?ys 0.0))
      else
        ;; first line theta is PI
        (assert (line 1 ?xs ?ys ?*PI*))
      )
      ;; second line penetrates elevator alcove
      (assert (line 2 850.0 0.0 ?*-HPI*))
```

```
    (if (< ?xg 850.0) then
      ;; theta for bline is PI
      (assert (bline ?xg ?yg ?*PI*))
    else
      ;; theta for bline is 0.0
      (assert (bline ?xg ?yg 0.0))
    )
  else
    ;; start in hall, end in hall: line, bline
    (if (< ?xs ?xg) then
      ;; first line theta is 0.0
      (assert (line 1 ?xs ?ys 0.0))
    else
      ;; first line theta is PI
      (assert (line 1 ?xs ?ys ?*PI*))
    )
    (if (< ?ys ?yg) then
      ;; bline theta is HPI
      (assert (bline ?xg ?yg ?*HPI*))
    else
      ;; bline theta is -HPI
      (assert (bline ?xg ?yg ?*-HPI*))
    )
  )
else
  ;; start is in elevator area
  (if (< ?yg 0.0) then
    ;; start in elevator, end in elevator: line, bline
    (if (< ?xs ?xg) then
      ;; first line theta is 0.0
      (assert (line 1 ?xs ?ys 0.0))
    else
      ;; first line theta is PI
      (assert (line 1 ?xs ?ys ?*PI*))
    )
    (if (< ?ys ?yg) then
      ;; bline theta is HPI
      (assert (bline ?xg ?yg ?*HPI*))
    else
      ;; bline theta is -HPI
      (assert (bline ?xg ?yg ?*-HPI*))
    )
  else
    ;; start in elevator, end in hall: line, line, bline
    ;; first line exits elevator area
    (assert (line 1 ?xs ?ys ?*HPI*))
    (if (< ?xs ?xg) then
      ;; second line theta is 0.0
      (assert (line 2 ?xs 120.0 0.0))
```

```
    else
      ;; second line theta is PI
      (assert (line 2 ?xs 120.0 ?*PI*))
    )
    (if (< 120.0 ?yg) then
      ;; bline theta is HPI
      (assert (bline ?xg ?yg ?*HPI*))
    else
      ;; bline theta is -HPI
      (assert (bline ?xg ?yg ?*-HPI*))
    )
  )
 )
)


;; ---------------------------------------
;;   Initiate left wall-following when an obstacle is encountered ahead
;;     in the direction the robot is trying to go, the goal is either
;;     ahead or to the right of the robot, and it is not already wall
;;     following. Compute line that crosses ahead to place obstacle on
;;     left side. Initiate right wall-following when the goal is to the
;;     right of the robot.
;;     Set stage value to zero in preparation for the situation where the
;;     hallway is blocked.

(deffunction decide-which-wall (?range ?xc ?yc ?tc ?xg ?yg ?tg)
 ;; need to decide if goal is to the left or to the right
 (if
  ;; robot facing north and xg < xc  or south and xg > xc
  ;;         or  west and yg < yc   or  east and yg > yc

    (or  (and (northp ?tc) (< ?xg ?xc))
         (and (southp ?tc) (> ?xg ?xc))
         (and (westp ?tc)  (< ?yg ?yc))
         (and (eastp ?tc)  (> ?yg ?yc))
    )

    ;; need to add test for presence of walls nearby, since that would
    ;; affect the decision to turn.

    then
      ;; initiate right wall-following
      (printout t "Started right wall follow " crlf )
      (assert (follow right wall =(sqdist ?xc ?yc ?xg ?yg)))
      (assert (started right wall-follow at ?xc ?yc))
      (assert (stage 0))
      (assert (turn interior left =(- ?range ?*obst-offset*)))
```

```
   else
      ;; initiate left wall-following
      (printout t "Started left wall follow " crlf )
      (assert (follow left wall =(sqdist ?xc ?yc ?xg ?yg)))
      (assert (stage 0))
      (assert (trun interior right =(- ?range ?*obst-offset*)))
   )
)


;; ---------------------------------------
;; Need to compute an x or y coordinate to wait until
;; when performing an exterior turn around a corner, this
;; ensures the robot will 'see' the wall when the sonar comes back on


(deffunction compute-wait-coordinate (?x ?y ?theta ?direction ?offset)
  (if (eq ?direction right) then
    (if (northp ?theta) then
      (assert (wait-until x GT =(+ ?x ?offset)))
    )
    (if (westp ?theta) then
      (assert (wait-until y GT =(+ ?y ?offset)))
    )
    (if (southp ?theta) then
      (assert (wait-until x LT =(- ?x ?offset)))
    )
    (if (eastp ?theta) then
      (assert (wait-until y LT =(- ?y ?offset)))
    )
  )
  (if (eq ?direction left) then
    (if (northp ?theta) then
      (assert (wait-until x LT =(- ?x ?offset)))
    )
    (if (westp ?theta) then
      (assert (wait-until y LT =(- ?y ?offset)))
    )
    (if (southp ?theta) then
      (assert (wait-until x GT =(+ ?x ?offset)))
    )
    (if (eastp ?theta) then
      (assert (wait-until y GT =(+ ?y ?offset)))
    )
  )
```

```
  (if (eq ?direction u-turn) then
    (if (northp ?theta) then
      (assert (wait-until y LT =(- ?y ?offset)))
    )
    (if (westp ?theta) then
      (assert (wait-until x GT =(+ ?x ?offset)))
    )
    (if (southp ?theta) then
      (assert (wait-until y GT =(+ ?y ?offset)))
    )
    (if (eastp ?theta) then
      (assert (wait-until x LT =(- ?x ?offset)))
    )
  )
)

;; ---------------------------------------------
;; Crude u-turn functions, issues four lines in sequence.
;; The first three are left (right) turns, the last one is a
;; right (left) turn, puts the robot back on original line, going
;; the other way.

(deffunction u-turn-left (?x ?y ?t)

    (bind ?new-x (+ ?x (* 20.0 (cos ?t))))
    (bind ?new-y (+ ?y (* 20.0 (sin ?t))))
    (bind ?new-theta (left-turn-func ?t ?*HPI*))
    (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
    (clips_line ?new-x ?new-y ?new-theta 0.0)

    (bind ?new-x (+ ?new-x (* 20.0 (cos ?new-theta))))
    (bind ?new-y (+ ?new-y (* 20.0 (sin ?new-theta))))
    (bind ?new-theta (left-turn-func ?new-theta ?*HPI*))
    (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
    (clips_line ?new-x ?new-y ?new-theta 0.0)

    (bind ?new-x (+ ?new-x (* 20.0 (cos ?new-theta))))
    (bind ?new-y (+ ?new-y (* 20.0 (sin ?new-theta))))
    (bind ?new-theta (left-turn-func ?new-theta ?*HPI*))
    (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
    (clips_line ?new-x ?new-y ?new-theta 0.0)

    (bind ?new-x (+ ?new-x (* 20.0 (cos ?new-theta))))
    (bind ?new-y (+ ?new-y (* 20.0 (sin ?new-theta))))
    (bind ?new-theta (right-turn-func ?new-theta ?*HPI*))
    (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
    (clips_line ?new-x ?new-y ?new-theta 0.0)
)
```

```
(deffunction u-turn-right (?x ?y ?t)

   (bind ?new-x (+ ?x (* 20.0 (cos ?t))))
   (bind ?new-y (+ ?y (* 20.0 (sin ?t))))
   (bind ?new-theta (right-turn-func ?t ?*HPI*))
   (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)

   (bind ?new-x (+ ?new-x (* 20.0 (cos ?new-theta))))
   (bind ?new-y (+ ?new-y (* 20.0 (sin ?new-theta))))
   (bind ?new-theta (right-turn-func ?new-theta ?*HPI*))
   (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)

   (bind ?new-x (+ ?new-x (* 20.0 (cos ?new-theta))))
   (bind ?new-y (+ ?new-y (* 20.0 (sin ?new-theta))))
   (bind ?new-theta (right-turn-func ?new-theta ?*HPI*))
   (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)

   (bind ?new-x (+ ?new-x (* 20.0 (cos ?new-theta))))
   (bind ?new-y (+ ?new-y (* 20.0 (sin ?new-theta))))
   (bind ?new-theta (left-turn-func ?new-theta ?*HPI*))
   (printout t "uturn line "?new-x" "?new-y" "?new-theta crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)
)
```

```
;;*****************************************
;; Filename:  clips.goal
;; Purpose:   Defines rules to be used for path planning.
;;            and verifying lines are safe.
;; Author:    Bob Fish
;;*****************************************


;; This rule starts the whole thing, works on initial-fact

(defrule plan-path-to-goal
  ?killit <- (initial-fact)
  (start-posit ?xs ?ys ?ts)
  (goal-posit ?xg ?yg ?tg)
 =>
  (printout t "Computing line(s) & bline" crlf)
  ;; Call originate-lines to issue appropriate lines.
  (originate-lines ?xs ?ys ?ts ?xg ?yg ?tg)
  (retract ?killit)
  (assert (examine-path))
)


;; ------------------------------------------------
;; This is where the path should be examined for conflict
;; with existing obstacles.  Since I am not implementing that
;; part, this is an automatic pass.

(defrule examine-path-for-safety
  ?killit <- (examine-path)
  ;; line(s) and blines created by originate-lines
 =>
  ;; call a function to check for impact with known obstacles
  ;; if okay, that function will assert path-okay fact
  (retract ?killit)
  (assert (path-okay))
)


;; ------------------------------------------------
;; Send the first line to the robot, calls clips_line
;; This needs a higher salience to ensure that it is issued
;; before the bline. Can't use a control fact, because
;; there might not be a line 2.

(defrule send-line-2-to-robot

  (declare (salience 10))
  (first line-sent)
  ?kill-line <- (line 2 ?xl ?yl ?tl)
 =>
```

```
  (printout t "Line 2 " ?xl ", " ?yl ", " ?tl crlf)
  (assert (old-line 2 ?xl ?yl ?tl))
  (retract ?kill-line)
  (clips_line ?xl ?yl ?tl 0.0)
)

;; ------------------------------------------
;; Now send the bline to the robot.
;; This rule asserts running fact, which starts the loop
;; that calls run_sim.

(defrule send-bline-to-robot
  (declare (salience 9))
  ?killit <- (first line-sent)
  ?kill-bline <- (bline ?xl ?yl ?tl)
  (goal-posit ?xg ?yg ?tg)
 =>
  (retract ?killit ?kill-bline)
  (printout t "Bline " ?xl ", " ?yl ", " ?tl crlf)
  (assert (old-bline  ?xl ?yl ?tl))
  (clips_bline ?xl ?yl ?tl 0.0)
  ;; (rotate to ?tg) should rotate to align with desired goal theta
  (assert (running))
  ;;  Assert dummy sonar and position facts to allow
  ;;  swap-sonar-and-position to fire the first time.
  (assert (sonar 0 0 0 0) (position 0 0 0))
)

;; ------------------------------------------------
;; Rules for re-issuing subset of original path after avoiding an obstacle.
;; This assumes that the first line is the one being intersected.

(defrule reissue-line1
  (new-path-planned)
  (old-line 1 ?xl ?yl ?tl)
 =>
  (printout t "old-line 1 " ?xl " " ?yl " " ?tl crlf)
  (clips_line ?xl ?yl ?tl 0.0)
  (assert (first old-line-sent))
)

;; salience 10 so this rule goes before bline
(defrule send-old-line-2-to-robot
  (declare (salience 10))
  (first old-line-sent)
  (old-line 2 ?xl ?yl ?tl)
 =>
  (printout t "old-line 2 " ?xl " " ?yl " " ?tl crlf)
  (clips_line ?xl ?yl ?tl 0.0)
)
```

```
(defrule send-old-bline-to-robot
  (declare (salience 9))
  ?killit <- (first old-line-sent)
  (old-bline ?xl ?yl ?tl)
  (goal-posit ?xg ?yg ?tg)
 =>
  (printout t "old-bline " ?xl " " ?yl " " ?tl crlf)
  (retract ?killit)
  (clips_bline ?xl ?yl ?tl 0.0)
  ;; need to rotate to align with goal theta. not implemented.
)
```

```
;;*****************************************
;;
;;  Filename:   clips.runner
;;  Purpose:    Defines rules to be used for operating the simulator
;;              by looping and calling run_sim.
;;  Author:     Bob Fish
;;*****************************************


;; The top of the loop is defined by rule run-sim, at salience 100, it
;; is the first rule to fire when its control fact is present.
;; The bottom of the loop is rule end-main-loop.  It retracts and
;; re-asserts the 'running' control fact. since its salience is -100,
;; any rules with higher salience, such as obstacle avoidance, will
;; get their chance to fire before the loop starts again.


(defrule run-sim
   (declare (salience 100))
   (running)
  =>
   (run_sim 2)
   (assert (swap-data))
)


(defrule new-sonar-and-position
  (declare (salience 90))
  ?killit  <- (sonar $?stuff)
  ?killone <- (position $?morestuff)
  ?killtwo <- (swap-data)
 =>
  (retract ?killit ?killone ?killtwo)
  (assert (=(get-vehicle)))
  (assert (=(get-sonar)))
)
```

```
;;*****************************************
;;Filename:   wallfoll.clp
;;Purpose:    Defines rules to be used for wall following,
;;            starting with detecting an obstacle, and
;;            ending with asking for the original lines to be
;;            reissued.
;;  Author:     Bob Fish
;;*****************************************


;;-----------------------------------------
;; A set of rules to detect obstacle ahead, right or left,
;; asserts blocked fact. Also companion rules to remove the
;; blocked fact if the obstacle is no longer in the way.


;; if either forward sonar detects, then obstacle ahead.
(defrule obstacle-ahead
  (declare (salience 15))
  (not (obstacle ahead))
  (sonar ?left ?aheadlft ?aheadrt ?right)
  (test (and (> ?aheadlft 9.3)
             (< ?aheadlft ?*obst-range*)
             (> (abs (- ?aheadlft (clips_wall_range 0 0.0))) 5.0)
        )

  )
 =>
  (printout t "blocked ahead" crlf)
  (assert (obstacle ahead))
)



(defrule remove-obstacle-ahead
  (declare (salience 15))
  ?killit <- (obstacle ahead)
  (not (sonar ?left
              ?aheadleft&:(and (> ?aheadleft 9.3) (< ?aheadleft ?*obst-range*))
              ?aheadrt
              ?right))
  (not (sonar ?left
              ?aheadleft
              ?aheadrt&:(and (> ?aheadrt 9.3) (< ?aheadrt ?*obst-range*))
              ?right))
 =>
  (retract ?killit)
  (printout t "no longer blocked ahead" crlf)
)
```

```
(defrule obstacle-right
  (declare (salience 15))
  (not (obstacle right))
  (sonar ?left ?ahead ?dummy ?right-range)
  (test (and (> ?right-range 9.3) (< ?right-range ?*obst-range*)))
 =>
  (printout t "blocked right" crlf)
  (assert (obstacle right))
)


(defrule remove-obstacle-right
  (declare (salience 15))
  ?killit <- (obstacle right)
  (not (sonar ?left
              ?ahead
              ?dummy
              ?right&:(and (> ?right 9.3) (< ?right ?*obst-range*))))
 =>
  (retract ?killit)
  (printout t "no longer blocked right" crlf)
)

(defrule obstacle-left
  (declare (salience 15))
  (not (obstacle left))
  (sonar ?left-range ?ahead ?dummy ?right)
  (test (and (> ?left-range 9.3) (< ?left-range ?*obst-range*)))
 =>
  (printout t "blocked left" crlf)
  (assert (obstacle left))
)


(defrule remove-obstacle-left
  (declare (salience 15))
  ?killit <- (obstacle left)
  (not (sonar ?left&:(and (> ?left 9.3) (< ?left ?*obst-range*))
              ?ahead
              ?dummy
              ?right))
 =>
  (retract ?killit)
  (printout t "no longer blocked left" crlf)
)
```

```
;; ------------------------------------------------
;; The first wall following rule. An obstacle has been detected
;; ahead, so call decide-which-wall to choose which way to turn.


(defrule need-to-wall-follow
  (obstacle ahead)
  (position ?xc ?yc ?tc)
  (goal-posit ?xg ?yg ?tg)
  (sonar ?left ?range-ahead ?dummy ?right)
  (not (follow ?any wall ?dist))
  (not (cannot-reach-goal))
 =>
  ;; call clips_flush to get rid of the pending motion commands
  (clips_flush)
  ;; call function to decide which way to follow
  (decide-which-wall ?range-ahead ?xc ?yc ?tc ?xg ?yg ?tg)
)

;; ------------------------------------------------
;; Rules to initiate an exterior turn or an interior turn, as
;; necessary while wall following.

(defrule right-wall-follow-exterior-corner
  (follow right wall ?dist)
  (not (obstacle right))
  (not (wait-until $?any))
 =>
  (assert (turn exterior right ?*corner-line-dist*))
  (assert (check-wall-follow-termination))
)

(defrule left-wall-follow-exterior-corner
  (follow left wall ?dist)
  (not (obstacle left))
  (not (wait-until $?any))
 =>
  (assert (turn exterior left ?*corner-line-dist*))
  (assert (check-wall-follow-termination))
)

(defrule right-wall-follow-interior-corner
  (follow right wall ?dist)
  (sonar ?left ?aheadlft ?dummy ?right)
  (position ?xc ?yc ?tc)
  (obstacle right)
  (obstacle ahead)
  (not (wait-until $?any))
 =>
  (assert (turn interior left =(- ?aheadlft ?*obst-offset*)))
)
```

```
(defrule left-wall-follow-interior-corner
  (follow left wall ?dist)
  (sonar ?left ?aheadlft ?dummy ?right)
  (position ?xc ?yc ?tc)
  (obstacle left)
  (obstacle ahead)
  (not (wait-until $?any))
 =>
  (assert (turn interior right =(- ?aheadlft ?*obst-offset*)))
)


;; ---------------------------------------------
;; Rules to reverse the direction of wall-following when the
;; robots sonar 0 gets within *REVERSAL-RANGE* of an environment
;; wall.  See function wall range.

(defrule reverse-right-wall-follow
  ?killit <-  (follow right wall ?dist)
  ?killtwo <- (stage ?num)
  (position ?x ?y ?t)
  (not (reversing))
  (test (< (abs (clips_wall_range 0 0.0)) ?*REVERSAL-RANGE*))

=>
  (printout t "reversing wall follow from right to left" crlf)
  (assert (follow left wall ?dist))
  (assert (reversing))
  (assert (stage =(+ ?num 1)))
  (retract ?killit ?killtwo)
  (u-turn-left ?x ?y ?t)
  (compute-wait-coordinate ?x ?y ?t u-turn ?*UTURN-DELAY*)
)

(defrule reverse-left-wall-follow
  ?killit <-  (follow left wall ?dist)
  ?killtwo <- (stage ?num)
  (position ?x ?y ?t)
  (not (reversing))
  (test (< (abs (clips_wall_range 0 0.0)) ?*REVERSAL-RANGE*))

=>
  (printout t "reversing wall follow from left to right")
  (assert (follow right wall ?dist))
  (assert (reversing))
  (assert (stage =(+ ?num 1)))
  (retract ?killit ?killtwo)
  (u-turn-right ?x ?y ?t)
  (compute-wait-coordinate ?x ?y ?t u-turn ?*UTURN-DELAY*)
)
```

```
;; ------------------------------------------------
;; Termination of wall following. Currently must be assisted by
;; manual assertion of the fact (okay).

(defrule terminate-wall-following
  (declare (salience 5))
  ?killone <- (follow ?any wall ?old-dist)
  ?killtwo <- (check-wall-follow-termination)
  (completed-turn)
  (okay)
  (goal-posit ?xg ?yg ?tg)
 =>
  (retract ?killone ?killtwo)
  (assert (reissue-sub-path))
  (printout t "Terminated wall following, need reissue remainder " crlf)
)


;; ------------------------------------------------------
;; Wall following has terminated, time to re-issue original path.

(defrule reissue-path
     ?killit <- (reissue-sub-path)
 =>

  (assert (new-path-planned))
  (retract ?killit)
)
```

```
;;*************************************
;; Filename:  clips.turn
;; Purpose:   Defines rules to be used for executing turns
;;            and specifying waiting period while turning.
;; Author:    Bob Fish
;;*************************************

;; ---------------------------------------------------
;; Right and left interior turns.  Also used when first starting
;; wall following.

(defrule right-interior-turn
   ?killit <- (turn interior right ?ahead)
   (position ?x ?y ?t)
   (not (wait-until $?any))
 =>
   (bind ?new-x (+ ?x (* ?ahead (cos ?t))))
   (bind ?new-y (+ ?y (* ?ahead (sin ?t))))
   (bind ?new-theta (right-turn-func ?t ?*HPI*))
   (printout t "New-line " ?new-x " " ?new-y " " ?new-theta crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)
   ;; for an interior turn, use theta to mark end of turn
   (bind ?theta-trip (+ ?new-theta ?*Beamwidth*))
   (assert (wait-until t LT ?theta-trip))
   (printout t "need to wait until t LT " ?theta-trip crlf)
   (retract ?killit)
)


(defrule left-interior-turn
   ?killit <- (turn interior left ?ahead)
   (position ?x ?y ?t)
   (not (wait-until $?any))
 =>
   (bind ?new-x (+ ?x (* ?ahead (cos ?t))))
   (bind ?new-y (+ ?y (* ?ahead (sin ?t))))
   (bind ?new-theta (left-turn-func ?t ?*HPI*))
   (printout t "new-line " ?new-x " " ?new-y " " ?new-theta crlf)
   ;; for an interior turn, use theta to mark end of turn
   (bind ?theta-trip (- ?new-theta ?*Beamwidth*))
   (assert (wait-until t GT  ?theta-trip))
   (printout t "need to wait until t GT " ?theta-trip crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)
   (retract ?killit)
)
```

```
;; ---------------------------------------------
;; Right and left exterior turn rules.

(defrule right-exterior-turn
   ?killit <- (turn exterior right ?ahead)
   (position ?x ?y ?t)
   (not (wait-until $?any))
 =>
   (bind ?new-x (+ ?x (* ?ahead (cos ?t))))
   (bind ?new-y (+ ?y (* ?ahead (sin ?t))))
   (bind ?new-theta (right-turn-func ?t ?*HPI*))
   (printout t "new-line " ?new-x " " ?new-y " " ?new-theta crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)
   (compute-wait-coordinate ?x ?y ?t right 53.0)
   (retract ?killit)
)

(defrule left-exterior-turn
   ?killit <- (turn exterior left ?ahead)
   (position ?x ?y ?t)
   (not (wait-until $?any))
 =>
   (bind ?new-x (+ ?x (* ?ahead (cos ?t))))
   (bind ?new-y (+ ?y (* ?ahead (sin ?t))))
   (bind ?new-theta (left-turn-func ?t ?*HPI*))
   (printout t "new-line " ?new-x " " ?new-y " " ?new-theta crlf)
   (clips_line ?new-x ?new-y ?new-theta 0.0)
   (compute-wait-coordinate ?x ?y ?t left 53.0)
   (retract ?killit)
)

;;---------------------------------------------
;; Two rules to watch for the point where robot's theta reaches
;; the desired value.


(defrule execute-wait-until-t-GT
    ?killmore <- (wait-until t GT ?val)
    (position ?x ?y ?t)
    (test (> ?t ?val))
 =>
    (printout t "matched wait until t GT " ?val crlf)
    (retract  ?killmore)
    (assert (completed-turn))
)


(defrule execute-wait-until-t-LT
    ?killmore <- (wait-until t LT ?val)
    (position ?x ?y ?t)
    (test (< ?t ?val))
```

```
 =>
    (printout t "matched wait until t LT " ?val crlf)
    (retract ?killmore)
    (assert (completed-turn))
)

;; ------------------------------------------------
;; Four rules to watch for the point where robot's x or y
;; coordinate reaches the desired value.

(defrule execute-wait-until-x-GT
    ?killmore <- (wait-until x GT ?val)
    (position ?x ?y ?t)
    (test (> ?x ?val))
 =>
    (printout t "matched wait until x GT " ?val crlf)
    (retract  ?killmore)
    (assert (completed-turn))
)

(defrule execute-wait-until-x-LT
    ?killone <- (wait-until x LT ?val)
    (position ?x ?y ?t)
    (test (< ?x ?val))
 =>
    (printout t "matched wait until x LT " ?val crlf)
    (retract  ?killone)
    (assert (completed-turn))
)
(defrule execute-wait-until-y-GT
    ?killmore <- (wait-until y GT ?val)
    (position ?x ?y ?t)
    (test (> ?y ?val))
 =>
    (printout t "matched wait until y GT " ?val crlf)
    (assert (completed-turn))
    (retract ?killmore)
)

(defrule execute-wait-until-y-LT
    ?killmore <- (wait-until y LT ?val)
    (position ?x ?y ?t)
    (test (< ?y ?val))
 =>
    (printout t "matched wait until y LT " ?val crlf)
    (retract ?killmore)
    (assert (completed-turn))
)
```

```
;; ---------------------------------------------------------
;; Need to eat the old completed-turn facts, use lower salience than
;; the terminate wall following rule,  so
;; terminate wall following gets first shot with completed-turn fact.

(defrule eat-completed-turn
  (declare (salience 3))
  ?killit <- (completed-turn)
 =>
  (retract ?killit)
)
```

APPENDIX D

THE EXPERT SYSTEM IN C

The Global definitions are from file mml.h

```
/* parameters for the fish expert system */
#define OBST_OFFSET    30.0  /* distance away from obstacle that
                        robot will wall follow at */
#define OBST_RANGE     60.0  /* sonar range at which an obstacle is
                        declared to be a danger.*/
#define BEAMWIDTH      .20   /* sonar half-beamwidth, used when
                        busy waiting for theta during turns */
#define THETA_ZERO     .780  /* If an angle is less than this, it is
                        considered zero. app PI/4 */
#define ENVLENGTH      2500.0 /* A number longer than the max length of
                         the environment, for line segment
                         formation  */

#define EXTERIOR_DELAY         50.0 /* How far around an exterior corner
                            the robot must travel before the
                            side sonars will regain the wall */
#define CORNER_TO_LINE_DIST    60.0 /* How far ahead the new line is on
                            exterior corners to end up with
                            the correct standoff distance from
                            the wall after the turn. Better to
                            compute this from a saved range.*/


typedef struct {
 POINT  begin,        /* beginning (tail) of a path segment */
       end;          /* end (arrowhead) of a path segment */
 double orientation;   /* orientation of path segment */
} PLANNED_PATH_SEGMENT;


typedef struct {
 int      ahead,   /* Boolean, set true when obstacle ahead */
          left,    /* Boolean, set true when obstacle left */
          right;   /* Boolean, set true when obstacle right */
} OBSTACLE_TYPE;

typedef struct {
 int      wfleft,  /* flag for wall following on the left */
          wfright, /* flag for wall following on the right */
          stage;   /* three stage process for blocked hallway */
```

```
} ROBOT_TYPE;


 /* instantiate some instances of the structures */

 PLANNED_PATH_SEGMENT orig_path_data[10];
 PLANNED_PATH_SEGMENT avoid_line;

 OBSTACLE_TYPE      obst_data;
 ROBOT_TYPE         robot_data;

 /* indexes for the orig_path_data array  */
 int first_orig_path_index;
 int last_orig_path_index;

/* the fish expert system declarations */
extern void plan_path_to_goal();
extern double right_turn_func();
extern double left_turn_func();
extern void turn_interior_left();
extern void turn_interior_right();
extern int northp();
extern int southp();
extern int eastp();
extern int westp();
extern void decide_which_wall();


/*****************************************************************
*****************************************************************
*****************************************************************
**
```

```
*****************************************************************
*****************************************************************
*****************************************************************
*****************************************************************
    USER'S Program                                  */


#include "mml.h"


user()
{
 CONFIGURATION start;
 CONFIGURATION goal;

 CONFIGURATION temporary;


 void initialize_structs();
 void turn_exterior_right();
 void turn_exterior_left();
 int  do_they_intersect();

 int intersect_path_index;  /*index to array */

 int i;                  /* loop variable */

 double s = 15.0;          /* size constant */

 /* turn on Declue debugger and initialize structures */
 enable_display_status();
 initialize_structs();


 /* Specify the start and goal positions */
 def_configuration(950.0, -450.0, HPI, 0.0, &start);
 def_configuration(380.0, 90.0,  PI, 0.0, &goal);

 /* turn on sonars */
 enable_sonar(0);
 enable_sonar(7);
 enable_sonar(4);

 /* set up robot, and call plan_path_to_goal  */
 size_const(s);
 speed(15.0);

 set_rob(&start);

 plan_path_to_goal(&start, &goal);
 r_printf("\npath computed\n");
```

```
/* loop to print stored line segments for troubleshooting */
/*  for (i=first_orig_path_index; i<last_orig_path_index+1; i++)
   {
r_printf ("\nbegin.x0=> ");
r_printfr(orig_path_data[i].begin.x0,2);
r_printf (" y0=> ");
r_printfr(orig_path_data[i].begin.y0,2);
r_printf (" th=> ");
r_printfr(orig_path_data[i].orientation,2);
r_printf ("\n  end.x0=> ");
r_printfr(orig_path_data[i].end.x0,2);
r_printf (" y0=> ");
r_printfr(orig_path_data[i].end.y0,2);
}  */



/*  start the big do-while loop, exit when the robot reaches the goal */
do
{


 /* First priority is to update the obstacle structures,
    depending on how close the obstacles are */
 /*-------------------------------------------------------*/
 if (sonar(0) > 9.3 && sonar(0) < OBST_RANGE)
   obst_data.ahead = TRUE;
 else
   obst_data.ahead = FALSE;

 if (sonar(7) > 9.3 && sonar(7) < OBST_RANGE)
  obst_data.right = TRUE;
 else
   obst_data.right = FALSE;

 if (sonar(4) > 9.3 && sonar(4) < OBST_RANGE)
   obst_data.left = TRUE;
 else
   obst_data.left = FALSE;



 /* The rest of the loop examines the structures for patterns
    that correspond to the CLIPS rules they were derived from.
    only one if statement will fire each time through. */

 /*-------------------------------------------------------*/
```

```
/* Not wall_following, and obstacle ahead detected */
if (obst_data.ahead && !(robot_data.wfleft || robot_data.wfright))
{

 /* Obstacle ahead, need to flush, start wall following */
 flush();

 r_printf("\nObstacle ahead, flush commanded\n");

 decide_which_wall(sonar(0), &goal);

}

/*-------------------------------------------------------*/
/* Wall following on the right, exterior corner detected */
else if (robot_data.wfright &&  !obst_data.right)
   {

 r_printf("\n exterior right corner detected\n");
 turn_exterior_right(CORNER_TO_LINE_DIST);

 /* When returns from the function call, the turn is complete.
    Need to check to see if  new line intercepts original path.
    Should also check to see if no obstacle in sonar range.*/

 intersect_path_index = do_they_intersect();

 if (intersect_path_index > -1 )
       {
  /* The avoidance line and the original path intersect,
     at index number intersect_path_index, now wall following
     can be terminated and the original path reissued starting
     with segment number intersect_path_index  */

  robot_data.wfright = 0;

  for (i=intersect_path_index; i<last_orig_path_index; i++)
          {
    def_configuration(orig_path_data[i].begin.x0,
                orig_path_data[i].begin.y0,
                orig_path_data[i].orientation,
                0.0, &temporary);

    line (&temporary);

  } /* end loop */
```

```
   /* Now issue the last original path as a bline */
   def_configuration(orig_path_data[last_orig_path_index].end.x0,
                     orig_path_data[last_orig_path_index].end.y0,
                     orig_path_data[last_orig_path_index].orientation,
                     0.0, &temporary);

   bline (&temporary);

        } /* end if(intersect_path_index > -1)  */

   }  /* end if exterior turn needed */

/*-------------------------------------------------------*/
/* Wall following on the left, exterior corner detected, similar action*/
else if (robot_data.wfleft &&  !obst_data.left)
   {

  r_printf("\nExterior left corner detected\n");
  turn_exterior_left(CORNER_TO_LINE_DIST);

  /* When returns from the function call, the turn is complete.
     Need to check to see if  new line intercepts original path.
     Should also check to see if no obstacle in sonar range.*/

  intersect_path_index = do_they_intersect();

  if (intersect_path_index > -1 )
        {
   /* The avoidance line and the original path intersect,
      at index number intersect_path_index, now wall following
      can be terminated and the original path reissued starting
      with segment number intersect_path_index  */

   robot_data.wfleft = 0;

   for (i=intersect_path_index; i<last_orig_path_index; i++)
           {
    def_configuration(orig_path_data[i].begin.x0,
                      orig_path_data[i].begin.y0,
                      orig_path_data[i].orientation,
                      0.0, &temporary);

    line (&temporary);

   } /* end loop */
```

```
    /* Now issue the last original path as a bline */
    def_configuration(orig_path_data[last_orig_path_index].end.x0,
                orig_path_data[last_orig_path_index].end.y0,
                orig_path_data[last_orig_path_index].orientation,
                0.0, &temporary);

    bline (&temporary);

        } /* end if(intersect_path_index > -1)  */
    }


/*-------------------------------------------------------*/
/* Wall following on the left, interior corner detected */
else if (robot_data.wfleft && obst_data.ahead)
    {
  r_printf("\nLeft wf: interior corner detected\n");
  turn_interior_right(sonar(0) - OBST_OFFSET);
    }

/*-------------------------------------------------------*/
/*Wall following on the right, interior corner detected */
else if (robot_data.wfright && obst_data.ahead)
    {
  r_printf("\n right wf: interior corner detected\n");
  turn_interior_left(sonar(0) - OBST_OFFSET);
    }


 /* Printout of sonar ranges whil in main loop, sometiemes the
    wait_timer acutally gives better results after a turn.
    printed for monitoring only. */
 r_printf ("\n sonar(0) => ");
 r_printfr(sonar(0),2);
 r_printf ("  ");
 r_printf (" sonar(7) => ");
 r_printfr(sonar(7),2);
 r_printf ("  ");
 r_printf (" sonar(4) => ");
 r_printfr(sonar(4),2);
 wait_timer(50);


}while (status != SSTOP); /* end while loop.Status will only go to
                    stop if the bline image reaches the goal */


r_printf("\I made it to the goal ");


}  /* end user */
```

```
/*********************************************************
*********************************************************
*********************************************************/


/
*************************************************************************
*******
 FUNCTION:   initialize_structs()
 PARAMETERS: void
 PURPOSE:    sets initial data in fields of structures
 RETURNS:
 CALLED BY:  user
 CALLS:
 COMMENTS:   18 Jun 93 -- Bob Fish
*************************************************************************
*******/

void initialize_structs()
{
 /*initialize the flags and fields in structures */
 obst_data.ahead    = FALSE;
 obst_data.left     = FALSE;
 obst_data.right    = FALSE;

 robot_data.wfleft  = FALSE;
 robot_data.wfright = FALSE;
 robot_data.stage   = -1;

}/* end initialize_structs */



/
*************************************************************************
*******
 FUNCTION:   do_they_intersect()
 PARAMETERS: void
 PURPOSE:    checks avoid_line and path array to see if wall following criteria
             are met.
 RETURNS:    int (True or false)
 CALLED BY:  user
 CALLS:
 COMMENTS:   21 Jun 93 -- Bob Fish
*************************************************************************
*******/
```

```
int do_they_intersect()

{

 /* This is a complicated Function.
    Plan:  Step through the array  orig_path_data,
           from first_orig_path_index to the
           last_orig_path_index, testing each line
           segment for intersection with avoid_line.

           If intersection occurs, (testing should be done
            to verify distance along path, see thesis)
           return the index that the intersection occurs at.

           If none of the line segments intersect, then
           return -1.  */

  int path_segments_cross();

  int j;

  for (j=first_orig_path_index; j<last_orig_path_index+1; j++)
     {
    if(path_segments_cross(orig_path_data[j], avoid_line))
          {
     r_printf("\nIntersection worked, at path number => ");
     r_printfi(j);

     return j;
     }

  } /* end loop */

  /* if reached this point then none of the segments crossed.
     time to return -1 */

  return -1;


 } /* end do_they_intersect  */


/
*************************************************************************
*******
 FUNCTION:   path_segments_cross()
 PARAMETERS: PLANNED PATH SEGMENT target, line
 PURPOSE:    checks two line segments to see if they cross, or touch
 RETURNS:    int (True or false)
 CALLED BY:  do_they_intersect
 CALLS:      cardinal headingp's,
```

```
 COMMENTS:  21 Jun 93 -- Bob Fish
*************************************************************************
*******/

int path_segments_cross(target, test_line)
PLANNED_PATH_SEGMENT target, test_line;
{

 /* this function is currently hardwired to succeed only for specific
    original path/avoidance arrangements. need to get line segment routines
    in here. See utilities.c, and similar stuff in serve_sonar(Simulator
    version, and wall_range in simulator. */

 if (northp(vehicle.t) && westp(target.orientation))
  return TRUE;
 else
  return FALSE;

}

/
*************************************************************************
*******
 FUNCTION:  northp(), southp(), westp(), and eastp()
 PARAMETERS: double heading
 PURPOSE:   determines if the input angle is near the approriate
            cardinal heading.
 RETURNS:   int (True or false)
 CALLED BY:  various in expert system
 CALLS:
 COMMENTS:  18 Jun 93 -- Bob Fish
*************************************************************************
*******/

int northp (heading)
 double heading;
{
 if (fabs (norm (heading - HPI)) < THETA_ZERO)
   return TRUE;
  else
   return FALSE;
}

int southp (heading)
 double heading;
{
 if (fabs (norm (heading - (3.0*HPI))) < THETA_ZERO)
   return TRUE;
  else
   return FALSE;
}
```

```
int westp (heading)
 double heading;
{
  if (fabs (norm (heading - PI)) < THETA_ZERO)
    return TRUE;
   else
    return FALSE;
}

int eastp (heading)
 double heading;
{
  if (fabs (norm (heading)) < THETA_ZERO)
    return TRUE;
   else
    return FALSE;
}


/
*************************************************************************
*******
 FUNCTION:   decide_which_wall()
 PARAMETERS: double range, CONFIGURATION  *gl
 PURPOSE:    Determines which is the best way to turn when faced
             with an obstacle ahead, based on goal position.
 RETURNS:    void, issues line command indirectly
 CALLED BY:  user
 CALLS:      northp, southp, westp, eastp,turn_interior_right/left
 COMMENTS:   11 Jun 93 -- Bob Fish
*************************************************************************
*******/

void decide_which_wall (range, gl)
double range;
CONFIGURATION *gl;
{

  CONFIGURATION goal;

  goal = *gl;

  /* First need to decide if goal is to the left or to the right
     if robot facing north and xg < xc or  south and xg > xc
               or  west and yg < yc  or  east and yg > yc then
     goal is to the robot's left.*/


  if( (northp(vehicle.t)&& (goal.x < vehicle.x)) ||
      (southp(vehicle.t)&& (goal.x > vehicle.x)) ||
```

```
    (westp (vehicle.t)&& (goal.y < vehicle.y)) ||
    (eastp (vehicle.t)&& (goal.y > vehicle.y)) )
        {
    /* This is the point to test for presence of nearby walls. don't
       want to turn in the direction of a wall if it is already close. */

    /* Initiate right-wall-following */
    r_printf("\nStarted right wall following ");

    robot_data.wfright = TRUE;
    robot_data.stage = 0;

    /* Call function to command left turn */
    turn_interior_left(range - OBST_OFFSET);
   }
   else
   {
    /* Initiate left wall-following */
    r_printf("\nStarted left wall following ");

    robot_data.wfleft = TRUE;
    robot_data.stage = 0;

    /* Call function to command right turn */
    turn_interior_right(range - OBST_OFFSET);
        } /* end if */

}/* end decide_which_wall */



/
**************************************************************************
*******
 FUNCTION:   left_turn_func(), right_turn_func()
 PARAMETERS: double orig, delta
 PURPOSE:    returns the appropriate new heading, turning from aligned
             old heading by amount delta.(normally HPI)
 RETURNS:    double, new heading
 CALLED BY:  turning command functions.
 CALLS:      align
 COMMENTS:   11 Jun 93 -- Bob Fish
**************************************************************************
*******/


double right_turn_func (orig, delta)
 double orig, delta;
{
  return ((align(orig) - delta));
}
```

```
double left_turn_func (orig, delta)
 double orig, delta;
{
  return ((align(orig) + delta));
}


/
**************************************************************************
*******
 FUNCTION:   turn_interior_left(), turn_interior_right()
 PARAMETERS: double range
 PURPOSE:    Computes configuration for a left or right turn
          at a point 'range' units ahead.computes wait until for theta.
 RETURNS:    void, issues line command
 CALLED BY:  wall following functions
 CALLS:      line, left/right_turn_func,
 COMMENTS:   11 Jun 93 -- Bob Fish
**************************************************************************
*******/

void turn_interior_left(range)
 double range;
{
  double left_turn_func();

  CONFIGURATION tempconf;

  double new_x, new_y, new_t;

  /* Compute coordinates of point 'range' units ahead of robot,
     with theta equal to a left turn. */
  new_x = vehicle.x + (range * cos(vehicle.t));
  new_y = vehicle.y + (range * sin(vehicle.t));
  new_t = left_turn_func(vehicle.t, HPI);

  /* Issue the new line command */
  def_configuration (new_x, new_y, new_t, 0.0, &tempconf);
  line(&tempconf);

  /* Wait until theta is greater than computed value so sonars can
     regain the target after the turn.  I could not get Sols wait_until
     to work when using theta.  */

  r_printf("\nstart wait_until\n");
  while(vehicle.t < (new_t - BEAMWIDTH));/* Busy loop */
  r_printf("\nCompleted wait_until\n");
 }
```

```
void turn_interior_right(range)
 double range;
{
 double right_turn_func();

 CONFIGURATION tempconf;

 double new_x, new_y, new_t;

 /* Compute coordinates of point 'range' units ahead of robot,
    with theta equal to a rightt turn. */
 new_x = vehicle.x + (range * cos(vehicle.t));
 new_y = vehicle.y + (range * sin(vehicle.t));
 new_t = right_turn_func(vehicle.t, HPI);

 /* Issue the new line command */
 def_configuration (new_x, new_y, new_t, 0.0, &tempconf);
 line(&tempconf);

 /* Wait until theta is greater than computed value so sonars can
    regain the target after the turn.  I could not get Sols wait_until
    to work when using theta.  */

 r_printf("\nstart wait_until\n");
 while(vehicle.t > (new_t + BEAMWIDTH)); /* Busy loop */
 r_printf("\nCompleted wait_until\n");
}


/
*************************************************************************
*******
 FUNCTION:   turn exterior right or left
 PARAMETERS: double ahead
 PURPOSE:    orders right or left turn around an exterior corner. Initiates
         wait_until call.
 RETURNS:    void
 CALLED BY:  user
 CALLS:      right/left turn_func, wait_until
 COMMENTS:   21 Jun 93 -- Bob Fish
*************************************************************************
*******/

void turn_exterior_right(ahead)
 double ahead;
{

 CONFIGURATION tempconf;
 double new_x, new_y, new_t;
```

```
new_x = vehicle.x + (ahead * cos(vehicle.t));
new_y = vehicle.y + (ahead * sin(vehicle.t));
new_t = right_turn_func(vehicle.t, HPI);

/* Trouble shooting print statements */
/*  r_printf ("\n Exterior line x=> ");
r_printfr(new_x, 2);
r_printf (" y => ");
r_printfr(new_y, 2);
r_printf (" t => ");
r_printfr(new_t, 2);  */

/* This new line is a candidate for intersection of a segment
   of the original path.  Save the data as a line segment */

avoid_line.begin.x0 = new_x;
avoid_line.begin.y0 = new_y;
avoid_line.orientation = new_t;

/* Stretch the line segment out for a long ways,
   to make sure it will intersect if possible, use  ENVLENGTH */

avoid_line.end.x0 = new_x + ENVLENGTH * cos(new_t);
avoid_line.end.y0 = new_y + ENVLENGTH * sin(new_t);

/* More monitoring print statements */
/*  r_printf ("\nbegin.x0=> ");
r_printfr(avoid_line.begin.x0,2);
r_printf (" y0=> ");
r_printfr(avoid_line.begin.y0,2);
r_printf (" th=> ");
r_printfr(avoid_line.orientation,2);
r_printf ("\n  end.x0=> ");
r_printfr(avoid_line.end.x0,2);
r_printf (" y0=> ");
r_printfr(avoid_line.end.y0,2);

wait_timer (300);  */

/* Issue the new line to turn the corner */
def_configuration(new_x, new_y, new_t, 0.0, &tempconf);
line(&tempconf);

/* Need to delay until corner is turned. */
r_printf("\nstart wait_until\n");

if (northp(vehicle.t))
   {
  /* need to wait until x GT */
  wait_until(X, GT, new_x + EXTERIOR_DELAY);
}
```

```
 else if (eastp(vehicle.t))
    {
  /* need to wait until y LT */
  wait_until(Y, LT, new_y - EXTERIOR_DELAY);
 }
 else if (southp(vehicle.t))
    {
  /* need to wait until x LT */
  wait_until(X, LT, new_x - EXTERIOR_DELAY);
 }
 else if (westp(vehicle.t))
    {
  /* need to wait until y GT */
  wait_until(Y, GT, new_y + EXTERIOR_DELAY);
 }

 r_printf("\nCompleted wait_until\n");


} /* end exterior_turn_right */



void turn_exterior_left(ahead)
 double ahead;
{

  CONFIGURATION tempconf;
  double new_x, new_y, new_t;

  new_x = vehicle.x + (ahead * cos(vehicle.t));
  new_y = vehicle.y + (ahead * sin(vehicle.t));
  new_t = left_turn_func(vehicle.t, HPI);

  /* Trouble shooting print statements */
  /* r_printf ("\n Exterior line x=> ");
  r_printfr(new_x, 2);
  r_printf (" y => ");
  r_printfr(new_y, 2);
  r_printf (" t => ");
  r_printfr(new_t, 2);  */

  /* This new line is a candidate for intersection of a segment
     of the original path.  Save the data as a line segment */

  avoid_line.begin.x0 = new_x;
  avoid_line.begin.y0 = new_y;
  avoid_line.orientation = new_t;

  /* Stretch the line segment out for a long ways,
     to make sure it will intersect if possible, use  ENVLENGTH */
```

```
avoid_line.end.x0 = new_x + ENVLENGTH * cos(new_t);
avoid_line.end.y0 = new_y + ENVLENGTH * sin(new_t);

/* More monitoring print statements */
/* r_printf ("\nbegin.x0=> ");
r_printfr(avoid_line.begin.x0,2);
r_printf (" y0=> ");
r_printfr(avoid_line.begin.y0,2);
r_printf (" th=> ");
r_printfr(avoid_line.orientation,2);
r_printf ("\n  end.x0=> ");
r_printfr(avoid_line.end.x0,2);
r_printf (" y0=> ");
r_printfr(avoid_line.end.y0,2);
wait_timer (300);  */


/* Issue the new line to turn the corner */
def_configuration(new_x, new_y, new_t, 0.0, &tempconf);
line(&tempconf);


/* Now compute the x or y coordinate to wait until,
   so that sonar will reacquire the wall after turning.*/

r_printf("\nstart wait_until\n");
if (northp(vehicle.t))
    {
  /* need to wait until x LT */
  wait_until(X, LT, new_x - EXTERIOR_DELAY);
 }
 else if (eastp(vehicle.t))
    {
  /* need to wait until y GT */
  wait_until(Y, GT, new_y + EXTERIOR_DELAY);
 }
 else if (southp(vehicle.t))
    {
  /* need to wait until x GT */
  wait_until(X, GT, new_x + EXTERIOR_DELAY);
 }
 else if (westp(vehicle.t))
    {
  /* need to wait until y LT */
  wait_until(Y, LT, new_y - EXTERIOR_DELAY);
 }

 r_printf("\nCompleted wait_until\n");

} /* end exterior_turn_left */
```

```
/
*************************************************************************
*******
 FUNCTION:   plan_path_to_goal()
 PARAMETERS: CONFIGURATION *st, *gl
 PURPOSE:    Calculate and issue line commands for path planning.
             Fish's expert system.
 RETURNS:    void
 CALLED BY:  user
 CALLS:      motion commands,
 COMMENTS:   11 Jun 93 -- Bob Fish
*************************************************************************
*******/


void plan_path_to_goal (st,gl)
CONFIGURATION *st;
CONFIGURATION *gl;
{
 CONFIGURATION start, goal, tempconf;

 double xtemp, ytemp, thtemp;

 start = *st;
 goal  = *gl;


 /* This is just a huge if then complex, analyzing the possibilities,
 and issuing lines that should work in the hallway. At the same time,
 load the line segments into 'orig_path_data' array for use when
 terminating wall following. */

 if (start.y >  20.0)
 {
  if (goal.y < 20.0)
       {
    /* start in hall, end in elevator, line, line, bline*/
    if (start.x <  850.0)
    {
     /*first line theta = 0.0 */
     thtemp = 0.0;
    }
    else
    {
     /* first line theta is PI*/
     thtemp = PI;
    }
```

```
def_configuration(start.x, start.y, thtemp, 0.0, &tempconf);
/* need to rotate */
line(&tempconf);

orig_path_data[5].begin.x0 = start.x;
orig_path_data[5].begin.y0 = start.y;
orig_path_data[5].orientation = thtemp;

/* second line penetrates elevator alcove */
def_configuration(850.0, 0.0, -HPI, 0.0, &tempconf);
line(&tempconf);

orig_path_data[5].end.x0 = 850.0;
orig_path_data[5].end.y0 = start.y;

first_orig_path_index = 5;

orig_path_data[6].begin.x0 = 850.0;
orig_path_data[6].begin.y0 = start.y;

if (goal.x < 850.0)
{
  /* theta for bline is PI */
  thtemp = PI;
}
else
{
  /* theta for bline is 0.0  */
  thtemp = 0.0;
}

/* do a bline to the goal */
def_configuration(goal.x, goal.y, thtemp, 0.0, &tempconf);
bline(&tempconf);

orig_path_data[6].end.x0 = 850.0;
orig_path_data[6].end.y0 = goal.y;
orig_path_data[6].orientation = -HPI;

orig_path_data[7].begin.x0 = 850.0;
orig_path_data[7].begin.y0 = goal.y;

orig_path_data[7].end.x0 = goal.x;
orig_path_data[7].end.y0 = goal.y;
orig_path_data[7].orientation = thtemp;

last_orig_path_index = 7;

}
else
```

```
{
 /* start in hall, end in hall: line, bline */
 if (start.x < goal.x)
 {
  /* first line theta is 0.0 */
  thtemp = 0.0;
 }
 else
 {
  /*  first line theta is PI  */
  thtemp = PI;
 }

 def_configuration(start.x, start.y, thtemp, 0.0, &tempconf);
 /* need to rotate */
 line(&tempconf);

 orig_path_data[5].begin.x0 = start.x;
 orig_path_data[5].begin.y0 = start.y;
 orig_path_data[5].orientation = thtemp;

 if (start.y < goal.y)
 {
  /* bline theta is HPI  */
  thtemp = HPI;
 }
 else
 {
  /* bline theta is -HPI */
  thtemp = -HPI;
 }

 /* do a bline to the goal */
 def_configuration(goal.x, goal.y, thtemp, 0.0, &tempconf);
 bline(&tempconf);

 orig_path_data[5].end.x0 = goal.x;
 orig_path_data[5].end.y0 = start.y;

 first_orig_path_index = 5;

 orig_path_data[6].begin.x0 = goal.x;
 orig_path_data[6].begin.y0 = start.y;
 orig_path_data[6].end.x0 = goal.x;
 orig_path_data[6].end.y0 = goal.y;
 orig_path_data[6].orientation = thtemp;

 last_orig_path_index = 6;

 }
}
```

```
else
{
 /* start is in elevator area */
 if (goal.y <  0.0) /* should this be 20.0 ? */
 {
  /* start in elevator, end in elevator: line, bline */
  if (start.x < goal.x)
  {
   /* first line theta is 0.0 */
   thtemp = 0.0;
  }
  else
  {
   /* first line theta is PI */
   thtemp = PI;
  }

  def_configuration(start.x, start.y, thtemp, 0.0, &tempconf);
  /* need to rotate */
  line(&tempconf);

  orig_path_data[5].begin.x0 = start.x;
  orig_path_data[5].begin.y0 = start.y;
  orig_path_data[5].orientation = thtemp;

  if (start.y < goal.y)
  {
   /* bline theta is HPI */
   thtemp = HPI;
  }
  else
  {
   /*  bline theta is -HPI  */
   thtemp = -HPI;
  }

  /* do a bline to the goal */
  def_configuration(goal.x, goal.y, thtemp, 0.0, &tempconf);
  bline(&tempconf);

  orig_path_data[5].end.x0 = goal.x;
  orig_path_data[5].end.y0 = start.y;

  first_orig_path_index = 5;

  orig_path_data[6].begin.x0 = goal.x;
  orig_path_data[6].begin.y0 = start.y;

  orig_path_data[6].end.x0 = goal.x;
  orig_path_data[6].end.y0 = goal.y;
  orig_path_data[6].orientation = thtemp;
```

```
 last_orig_path_index = 6;

 }
else
{
 /* start in elevator, end in hall: line, line, bline
    first line exits elevator area  */

 def_configuration(start.x, start.y, HPI, 0.0, &tempconf);
 /* need to rotate */
 line(&tempconf);

 orig_path_data[5].begin.x0 = start.x;
 orig_path_data[5].begin.y0 = start.y;
 orig_path_data[5].orientation = HPI;

 if (start.x < goal.x)
 {
  /* second line theta is 0.0  */
  thtemp = 0.0;
 }
 else
 {
  /* second line theta is PI  */
  thtemp = PI;
 }

 orig_path_data[5].end.x0 = start.x;
 orig_path_data[5].end.y0 = 120.0;

 first_orig_path_index = 5;

 orig_path_data[6].begin.x0 = start.x;
 orig_path_data[6].begin.y0 = 120.0;

 /* Second line goes down middle of hall */
 def_configuration(start.x, 120.0, thtemp, 0.0, &tempconf);
 line(&tempconf);

 orig_path_data[6].end.x0 = goal.x;
 orig_path_data[6].end.y0 = 120.0;
 orig_path_data[6].orientation = thtemp;

 orig_path_data[7].begin.x0 = goal.x;
 orig_path_data[7].begin.y0 = 120.0;


 if (goal.y > 120.0)
 {
  /*  bline theta is HPI  */
```

```
      thtemp = HPI;
     }
     else
     {
      /* bline theta is -HPI */
      thtemp = -HPI;
     }

     /* do a bline to the goal */
     def_configuration(goal.x, goal.y, thtemp, 0.0, &tempconf);
     bline(&tempconf);

     orig_path_data[7].end.x0 = goal.x;
     orig_path_data[7].end.y0 = goal.y;
     orig_path_data[7].orientation = thtemp;

     last_orig_path_index = 7;
    }
   }
} /* end of function plan_path_to_goal*/
```

LIST OF REFERENCES

[Alex93] Alexander, J. A., *Motion Control and Obstacle Avoidance for Autonomous Vehicles Using Simple Planar Curves*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1993.

[Brutz92] Brutzman, D. P., *NPS AUV Integrated Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1992.

[Giar91] Giarratano, Joseph C., *CLIPS User's Guide, Vols 1 & 2*, Software Technology Branch, NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

[Kana92] Kananyama, Y., "Introduction to Spatial Reasoning," Lecture Notes of the Advanced Robotoics Course, Dept. of Computer Science, Naval Postgraduate School, Spring Quarter, 1992.

[Kana93] Kanayama, Y., MacPherson, D., and Krahn, G., "Vehicle Motion Control and Analysis Using 2D Transformations," submitted to International Conference on Robotics and Automation, in Atlanta, GA, May 2-7 1993.

[MacPh93] MacPerson D. and Kanayama, Y., " A Navigation Algorithm for an Autonomous Vehicle with Real-Time Odometry Error Detection and Correction," submitted to International Conference on Robotics and Automation, in Atlanta, GA, May 2-7 1993.

[User93] Naval Postgraduate School, Monterey, CA Technical Report (Draft), Yamabico User's Manual, by D. MacPherson, R. Fish, and M. DeClue, p. 2, June 1993.

[Schmidt93] Schmidt, D. A., *NPSNET: A Graphical Based Expert System to Model P-3 Aircraft Interaction with Submarines And Ships*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1993.

[NASA91] Sherfey, S. R., *A Mobile Robot Sonar System*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

[NASA91] Software Technology Branch, *CLIPS Reference Manual, Vols I - III*, NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

Dudley Knox Library 2
Code 052
Naval Postgraduate School
Monterey, CA 93943-5002

Dr. Yutaka Kanayama 2
Computer Science Department Code CS/Ka
Naval Postgraduate School
Monterey, CA 93943

Dr. Anthony Healey 1
Mechanical Engineering Department Code ME/Hy
Naval Postgraduate School
Monterey, CA 93943

Dr. Sehung Kwak 1
Computer Science Department Code CS/Kw
Naval Postgraduate School
Monterey, CA 93943

LCDR Donald Brutzman 1
Operations Research Department Code OR/Br
Naval Postgraduate School
Monterey, CA 93943

LCDR Robert W. Fish 1
2217 First Landing Lane
Virginia Beach, VA 23451

Dr. Lincoln T. Fish 1
7 Osborne Rd.
Gorham, ME 04038